UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

Departamento de Arquitectura de Computadores y Automática

ANÁLISIS E IMPLEMENTACIÓN DE TÉCNICAS HARDWARE PARA ORGANIZACIONES DE LA MEMORIA DE INSTRUCCIONES DE BAJO CONSUMO DE ENERGÍA EN SISTEMAS EMPOTRADOS.
ANALYSIS AND IMPLEMENTATION OF HARDWARE TECHINIQUES FOR LOW-ENERGY INSTRUCTION MEMORY ORGANISATION IN EMBEDDED SYSTEMS

MEMORIA PARA OPTAR AL GRADO DE DOCTOR
PRESENTADA POR

Antonio Artés García

Bajo la dirección de los doctores

José L. Ayala Rodrigo
Francky Catthoor

MADRID, 2013

Análisis e Implementación de Técnicas Hardware para Organizaciones de la Memoria de Instrucciones de Bajo Consumo de Energía en Sistemas Empotrados

Analysis and Implementation of Hardware Techniques for Low-Energy Instruction Memory Organisations in Embedded Systems

Tesis Doctoral / Ph.D. Thesis

Antonio Artés García

Departamento de Arquitectura de Computadores y Automática
Facultad de Informática
Universidad Complutense de Madrid

Departement Elektrotechniek - ESAT
Faculteit Ingenieurswetenschappen
Katholieke Universiteit Leuven

2013

This document is ready to be printed double-sided.

Análisis e Implementación de Técnicas Hardware para Organizaciones de la Memoria de Instrucciones de Bajo Consumo de Energía en Sistemas Empotrados

Analysis and Implementation of Hardware Techniques for Low-Energy Instruction Memory Organisations in Embedded Systems

Tesis Doctoral presentada por / Ph.D. Thesis presented by

Antonio Artés García

Departamento de Arquitectura de Computadores y Automática
Facultad de Informática
Universidad Complutense de Madrid

Departement Elektrotechniek - ESAT
Faculteit Ingenieurswetenschappen
Katholieke Universiteit Leuven

2013

D/2013/7515/75

ISBN 978-94-6018-689-9

Análisis e Implementación de Técnicas Hardware para Organizaciones de la Memoria de Instrucciones de Bajo Consumo de Energía en Sistemas Empotrados

Memoria presentada por D. Antonio Artés García para optar al grado de Doctor en Ingeniería Informática por la Universidad Complutense de Madrid y al grado de Doctor en Engineering Science por la Katholieke Universiteit Leuven. Este trabajo se ha realizado bajo la dirección del Prof. Dr. José L. Ayala Rodrigo (Departamento de Arquitectura de Computadores y Automática, Universidad Complutense de Madrid) y Prof. Dr. Francky Catthoor (Department of Electrical Engineering, Katholieke Universiteit Leuven).

Madrid, julio de 2013.

Analysis and Implementation of Hardware Techniques for Low-Energy Instruction Memory Organisations in Embedded Systems

Dissertation presented by Mr. Antonio Artés García in order to apply for the degree of Doctor in Ingeniería Informática from Universidad Complutense de Madrid and for the degree of Doctor in Engineering Science from Katholieke Universiteit Leuven. This work has been supervised by Prof. Dr. José L. Ayala Rodrigo (Departamento de Arquitectura de Computadores y Automática, Universidad Complutense de Madrid) and Prof. Dr. Francky Catthoor (Department of Electrical Engineering, Katholieke Universiteit Leuven).

Madrid, July 2013.

Este trabajo ha sido posible gracias a la Comisión Interministerial de Ciencia y Tecnología, por las ayudas recibidas a través de los proyectos CICYT TIN2005/5619 y CICYT TIN2008/00508, al proyecto de investigación TEC2012-33892, y a la beca de investigación FPI BES-2009-023681.

A mi familia,
y a ti, suerte de mi vida.

"Pluralitas non est ponenda sine necessitate."
Occam's razor — William of Ockham.

Acknowledgements

"Nothing is more honourable than a grateful heart."

— Lucius Annaeus Seneca.

This doctoral thesis began as any great story. A young kid, facing a great opportunity to change of environment, finds himself at 6 am on a plane flying to an unknown country. A country called The Netherlands. That kid could never imagine what awaited him on his arrival in that country. What was in a first moment an internship of six months abroad, later changed to be an internship of 12 months, and finally, the idea of being a doctoral thesis appeared. Since then, several years of hard work, in which long meetings, round trips, unpublished/published articles, cold winters, and hot summers are included, have produced the greatest disappointments and the greatest illusions of the life of this kid. This period has been a moment of change after change in his life. However, before this kid realises, he has managed to complete the biggest project that he has had in his hands. Now, this kid, I mean ME, takes the opportunity that is provided by this special occasion to thank all the support and affection received through all these last years. Winds of change and fresh air are coming. However, in this special situation, in which I happily establish the end of this period of my life, I feel the obligation to remind all those people who not only have been around me during these wonderful years, but also have help me to achieve this project of my life. I would like especially to thank:

... from the heart to Francky Catthoor, because without your guidance and direction, this Ph.D. thesis would not have been possible. You have been for me the backbone of all my work, and the person that I have always resort both in good times and in bad times. Every conversation with you, in terms of either work or personal, has influenced both this doctoral thesis and me. Do not have the slightest doubt that every productivity, sports, leisure, and efficiency tip will always be present in both my personal and professional life. Thank you for the opportunity to work with you, and thank you for the chance to have known you.

... formally to my whole set of supervisors: Filipa Duarte, Maryam Ashouei, Jos Huisken, and José L. Ayala Rodrigo. Everything that you have taught me during this years of work will be really useful for my future.

... officially to *IMEC* and the people working for this company. Thanks for showing me how a real company works.

... friendly to Javier Gonzalo Ruiz, Christian Bachmann, Andreas Genser, and Jérôme Azémar. Javi, it was a pleasure to share and suffer my first days in The Netherlands with you. I will never forget our first and only "official" meeting together, as well our time at *Bierprofessor*. I am where I am because of you. Chris and Andy, thanks for being my eldest brothers during my first stay in Eindhoven and for having the enough patience to teach me English. Jérôme, thanks for the laughs and the party. ¡Joder, Hermano!.

... without words to Serdar Yildirim. Knowing and having a teacher of the life like you has been the best thing that ever happened to me. You took me in like a parent in The Netherlands and, with the passage of time, not only you have become a great friend, but also you have taught me the definition of friendship. If I have to point out the better, I would point out to you. Everyone deserves in her life someone like you.

... with words to Jef van de Molengraft. You always knew how to leave your trace in each person. Few people deserves my sincere thanks for helping me with this big project of my life, and you are among them. Your hospitality, your different view of life, your commitment in personal, working, and partying area, and your inside have contributed to dedicate you this doctoral thesis. For that, and for you. You will always be among us.

... literally to Katholieke Universiteit of Leuven and Universidad Complutense de Madrid to not only show me how different academic environments work, but also allow me being the first student to perform a joint doctorate between these two universities. At the end, all the papers, documents, reports, and forms have been worth.

... as a partner to my mates at Facultad de Informática of the Universidad Complutense de Madrid. I would like specially to mention Pablo García del Valle, due to his patience at the office and for his cooperation and assistance in day to day. The memory of the lunch time in the cafeteria of the university with all of you will always bring back fond memories.

... as a colleague to my former friends at Escuela Técnica Superior de Ingenieros de Telecomunicación of the Universidad Politécnica de Madrid for

being available to go out and share sorrows and joys. I would like specially to thank Ángel García Fernández for his comments and corrections between beers, for his insistence on going out and to air the mind, and for his support during the creation of this document.

... as a good son to my parents Encarna García Alonso and Antonio Artés Cantón. Sobre todo, gracias a vosotros, porque siempre habéis estado ahí. Gracias a vosotros soy quien soy, y por ello, he podido llegar hasta aquí. Gracias por el apoyo y por poner toda vuestra confianza en mí desde el primer día.

... as the eldest brother to my big brother Diego Onofre Artés García. He has brought sanity and wisdom to my days of work and he is the best applying *Risoterapia* in day to day. In addition, providing solutions, practical or not, he is the best. I wish him all the best for his life, and I hope he will be as proud of himself as I am of him.

... with love to Marta Sequeiros Fernández. I say it best when I say nothing at all. You are the only person that can understand this book, and because of that, this book is yours.

The doctoral thesis more than a written document is a lesson in life. Due to this fact, I must remember all those who have tried to place obstacles in my way. Thanks to them I have learnt to walk on risky and dangerous ground. For all of you, a simple "THANK YOU".

Finally, I cannot forget you, luck of my life, for giving me the persistence, the perseverance, and the willpower that I have been needing. Thanks to you, luck of my life for helping me in this long journey. I hope that you continue being present in my day to day, while I learn to fly.

"A dream you dream alone is only a dream. A dream you dream together is reality."

— John Winston Lennon.

Abstract

"I cannot do a summary of my life, because it is made up of various times and circumstances, books, friendships and fights, and that, only admits partial summaries."

— Carlos Monsiváis Aceves.

The design of current embedded systems is constrained by the requirements of modern embedded applications. Many of these applications require not only sustained operation for long periods of time, but also to be executed on battery powered systems. Under the constraint of not being mains-connected, the absence of wires to supply a constant source of energy causes that the use of an energy harvesting source or an integrated energy supplier limits the operation time of these electronic devices. Instruction memory organisations are pointed out as one of the major sources of energy consumption in embedded systems. As these systems are characterised by restrictive resources and a low-energy budget, any enhancement that is introduced in the instruction memory organisation allows not only to decrease the energy consumption, but also to have a better distribution of the energy budget throughout the embedded system. This Ph.D. thesis focuses on the study, analysis, proposal, implementation, and evaluation of low-energy optimisation techniques that can be used in the instruction memory organisations of embedded systems. Real-life embedded applications of the specific subdomain of wireless sensor nodes are used as benchmarks to show, analyse, and corroborate the benefits and disadvantages of each one of the concepts in which this Ph.D. thesis is based on. The first key contribution is the systematic study of existing low-energy optimisation techniques that are used in instruction memory organisations, outlining their comparative advantages, drawbacks, and trade-offs. On top of that, the experimental evaluation that is presented in this Ph.D. thesis uses a systematic method in order to have an accurate estimation of parasitics and switching activity. Due to this fact, this evaluation guides embedded systems designers to make the correct decision in the trade-offs that exist between energy budget, required performance, and area cost of the embedded system. The second key contribution is the development of a high-level energy estimation tool that, for a given

application and compiler, allows the exploration not only of architectural and compiler configurations, but also of code transformations that are related to the instruction memory organisation. The third key contribution is the proposal and analysis of several promising implementations of energy-efficient instruction memory organisations for a specific set of application codes and embedded architectures. Based on the previous contributions, the work that is presented in this Ph.D. thesis proves why further optimising instruction memory organisations from the energy consumption point of view will remain an extremely important trend in the future.

Keywords: Energy; Performance; Area; Design Space Exploration; Loop Buffer Architecture; Instruction Memory Organisation; Embedded System.

Samenvatting

"Ik kan niet een samenvatting van mijn leven, omdat het is samengesteld uit verschillende tijden en omstandigheden, boeken, vriendschappen en ruzies, en dat, ondersteunt alleen een gedeeltelijke samenvattingen."

— Carlos Monsiváis Aceves.

Het ontwerp van reële-tijd ingebedde systemen wordt beperkt door de eisen van de moderne ingebedde toepassingen. Veel van deze toepassingen vereisen niet alleen gebruik gedurende langere tijd, maar worden gevoed door batterijen. Die laatste hebben een beperkte opslag wat de energievoorziening beperkt maakt in duur. Dat heeft een grote impact op de haalbare bedrijfstijd van deze elektronische apparaten. Instructiegeheugen organisatie vormt een van de belangrijkste bronnen van energieverbruik in ingebedde systemen. Aangezien deze systemen heel energiezuinig moeten zijn, maakt elke verbetering die wordt geïntroduceerd in de instructiegeheugen organisatie de totale energieverbruik lager, maar dit laat ook toe om een betere verdeling te bekomen in de energiehuishouding van het ingebedde systeem. Dit Ph.D. proefschrift richt zich op de studie, analyse, voorstelling, implementatie, en evaluatie van technieken met lage energie die gebruikt kunnen worden in de organisaties van het instructiegeheugen van ingebedde systemen. Realistische toepassingen uit specifieke subdomeinen van draadloze sensorknopen worden als referentie gebruikt om de voor en nadelen van alle concepten aan te tonen, te analyseren en te bevestigen. De eerste belangrijke bijdrage is de systematische studie van de bestaande technieken die de energie verlagen en die gebruikt worden in bestaande geheugenorganisaties. Daarnaast maakt de experimentele evaluatie, gepresenteerd in dit Ph.D. proefschrift, gebruik van een systematische methode om een juiste schatting van alle bijdrages in de schakelactiviteit mee te nemen in het model. Daardoor kan de ontwerper van ingebedde systemen de juiste beslissing nemen in de afwegingen die bestaan tussen energiebudget, vereiste prestaties en andere kosten van het ingebedde systeem. De tweede belangrijke bijdrage is de ontwikkeling van een raamwerk voor de hoogniveau energieschattin, voor een bepaalde toepassing en compiler. Dit maakt de exploratie mogelijk van de architecturale en

compiler configuraties, maar ook van codetransformaties die gerelateerd zijn aan de instructiegeheugenorganisatie. De derde belangrijke bijdrage is het voorstel en analyse van een aantal veelbelovende implementaties van energie-efficiënte organisaties van het instructiegeheugen voor een specifieke set van toepassingen en ingebedde architectuurplatformen. Op basis van eerdere bijdrages, bewijst het werk gepresenteerd in dit Ph.D. proefschrift waarom verdere optimalisering van de instructiegeheugen organisatie vanuit het oogpunt van energieverbruik een uiterst belangrijke trend blijft in de toekomst.

Trefwoorden: Energie; Uitvoeringstijd; Gebied; Ontwerpruimte Exploratie; *Loop Buffer* Architectuur; Instructiegeheugen Organisatie; Ingebedde Systemen.

Resumen

"No puedo hacer un resumen de mi vida, porque está conformada por varias épocas y circunstancias, libros, amistades y pleitos, y eso, sólo admite resúmenes parciales."

— Carlos Monsiváis Aceves.

El diseño de los sistemas empotrados actuales se encuentra limitado por las exigencias de las aplicaciones empotradas modernas. Muchas de estas aplicaciones requieren no sólo un funcionamiento sostenido durante largos períodos de tiempo, sino también ser ejecutadas en sistemas alimentados por baterías. Bajo la restricción de no encontrarse alimentados por la red eléctrica principal, hace que la ausencia de cables para suministrar una fuente constante de energía provoque que el uso de una fuente de energía recolectora o de un proveedor de energía integrado limite el tiempo de funcionamiento de estos dispositivos electrónicos. La organización de la memoria de instrucciones está señalada como una de las principales fuentes de consumo de energía en los sistemas empotrados. Como estos sistemas se encuentran caracterizados por poseer recursos limitados y bajos consumos de energía, cualquier mejora que se introduzca en la organización de la memoria de instrucciones va a permitir no sólo reducir el consumo de energía, sino también mejorar la distribución de este consumo en todo el sistema empotrado. Esta tesis doctoral se centra en el estudio, análisis, propuesta, ejecución y evaluación de las técnicas de optimización de baja energía que se pueden utilizar en las organizaciones de la memoria de instrucciones de los sistemas empotrados. Aplicaciones empotradas de la vida real del subdominio específico de los nodos de sensores inalámbricos son usadas como *benchmarks* para mostrar, analizar y corroborar las ventajas y desventajas de cada uno de los conceptos en los que esta tesis doctoral se basa. La primera contribución clave es el estudio sistemático de las técnicas de optimización de bajo consumo actuales que se utilizan en las organizaciones de la memoria de instrucciones, destacando sus ventajas, desventajas y compromisos. Además de esto, la evaluación experimental que se presenta en esta tesis doctoral utiliza un método sistemático con el fin de tener una estimación exacta de las actividades parasitarias y de conmutación. Debido a este hecho, esta evaluación sirve de guía a los diseñadores de

sistemas empotrados para tomar la decisión correcta en los compromisos que existen entre el presupuesto de energía, rendimiento requerido, y el coste de área del sistema empotrado. La segunda contribución clave es el desarrollo de una herramienta de estimación de energía de alto nivel que, para una aplicación y compilador dados, permite la exploración no sólo de las configuraciones arquitectónicas y de compilador de la organización de la memoria de instrucciones, sino también de las transformaciones de código que se encuentran relacionadas con ésta. La tercera contribución clave es la propuesta y el análisis de varias implementaciones que son prometedoras en base a la eficiencia energética de la organización de la memoria de instrucciones para un conjunto específico de aplicaciones y sistemas empotrados. En base a las contribuciones anteriores, el trabajo que se presenta en esta tesis doctoral demuestra por qué la optimización de la organización de la memoria de instrucciones desde el punto de vista del consumo de energía seguirá siendo una tendencia muy importante en el futuro.

Palabras clave: Energía; Rendimiento; Área; Exploración del Espacio de Diseño; Arquitectura de *Loop Buffer*; Organización de la Memoria de Instrucciones; Sistema Empotrado.

Contents

List of Figures

List of Tables

List of Acronyms

AES ..... *Advanced Encryption Standard*

AG ..... *Address Generation Unit*

ALU ..... *Arithmetic Logic Unit*

ASIC ..... *Application-Specific Integrated Circuit*

ASIP ..... *Application-Specific Instruction-Set Processor*

BCLB ..... *Banked Central Loop Buffer Architecture*

BLB ..... *Banked Loop Buffer*

BTB ..... *Branch Target Buffer*

CELB ..... *Central Loop Buffer Architecture for Single Processor Organisation*

CLLB ..... *Clustered Loop Buffer Architecture with Shared Loop-Nest Organisation*

CM ..... *Constant Memory*

CMOS ..... *Complementary Metal-Oxide-Semiconductor*

CWT ..... *Continuous Wavelet Transform*

DFC ..... *Decoder Filter Cache*

DIB ..... *Decoded Instruction Buffer*

DLB ..... *Distributed Loop Buffer Architecture with Incompatible Loop-Nest Organisation*

DLP ..... *Data-Level Parallelism*

DM ..... *Data Memory*

DMH ..... *Data Memory Hierarchy*

DRAM ..... *Dynamic Random Access Memory*

DSP ..... *Digital Signal Processor*

DVLIW ..... *Distributed Very Long Instruction Word*

DWT ..... *Discrete Wavelet Transform*

ECG ..... *Electrocardiogram*

EEG ..... *Electroencephalogram*

FF ..... *Flip-Flop*

FFT ..... *Fast Fourier Transform*

FIFO ..... *First In, First Out*

FIPS ..... *Federal Information Processing Standard*

HBD ..... *Heartbeat Detection*

HBTC ..... *History-Based Tag-Comparison*

HDL ..... *Hardware-Description Language*

ICA ..... *Individual Component Analysis*

ILP ..... *Instruction-Level Parallelism*

IMO ..... *Instruction Memory Organisation*

IMOSIM ..... *Instruction Memory Organisation SIMulator*

IP ..... *Intellectual Property*

IQ ..... *Instruction Queue*

ISA ..... *Instruction Set Architecture*

ISS ..... *Instruction Set-Simulator*

ITRS ..... *International Technology Roadmap for Semiconductors*

JMD ..... *Jerarquía de la Memoria de Datos*

LB ..... *Loop Buffer*

LP ..... *Low Power*

LSD ..... *Loop Stream Detector*

MRFA ..... *Multi-Resolution Frequency Analysis*

MUL ..... *Multiplication Unit*

NIST ..... *National Institute of Standards and Technology*

NOLB ..... *NO Loop Buffer*

OMI ..... *Organización de la Memoria de Instrucciones*

OS ..... *Operating System*

PC ..... *Personal Computer*

PCA ..... *Principle Component Analysis*

PM ..... *Program Memory*

PVT ..... *Process, Voltage, and Temperature*

R ..... *Register File*

RISC ..... *Reduced Instruction Set Computer*

ROM ..... *Read Only Memory*

RTL ..... *Register-Transfer Level*

SDK ..... *Software Development Kit*

SH ..... *Shift Unit*

SIMD ..... *Single-Instruction Multiple-Data*

SMT ..... *Simultaneous Multi-Threading*

SoC ..... *System-on-Chip*

SPEC ..... *Standard Performance Evaluation Corporation*

SPM ..... *Scratchpad Memory*

SRAM ..... *Static Random Access Memory*

STFT ..... *Short-Time Fourier Transform*

TLP ..... *Thread-Level Parallelism*

TSMC ..... *Taiwan Semiconductor Manufacturing Company*

V ..... *Vector Register File*

VCD ..... *Value Change Dump*

VLIW ..... *Very Long Instruction Word*

VM ..... *Vector Memory*

WBAN..... *Wireless Body Area Network*

WSN..... *Wireless Sensor Network*

WWW..... *World Wide Web*

Chapter 1

Introduction

"Far out in the uncharted backwaters of the unfashionable end of the western spiral arm of the Galaxy lies a small unregarded yellow sun. Orbiting this at a distance of roughly ninety-two million miles is an utterly insignificant little blue green planet whose ape-descended life forms are so amazingly primitive that they still think digital watches are a pretty neat idea."

The Hitchhiker's Guide to the Galaxy
— Douglas Noel Adams (Douglas Adams).

This introductory Chapter presents the motivation and the problem context within which the research that is presented in this Ph.D. thesis resides. Besides, it is in this Chapter where the problem of the energy consumption of the instruction memory organisation in embedded systems is defined. Finally, this Chapter provides an overview of the content of the rest of this Ph.D. thesis.

1.1. Motivation and Context

The increasing use of battery powered systems has made the reduction of the energy consumption become an important design goal in the domain of embedded systems. However, it is necessary to deeply study the context of these systems in order to understand where the limit of this important design goal resides. Section 1.1.1 and Section 1.1.2 help the reader understand the context and the motivation of the work that is presented in this Ph.D. thesis.

1.1.1. Embedded Systems

The rapid improvement that computer technology has suffered in the last decades has come both from advances in the technology used to build

computers and from innovation in computer design. Although technological improvements have been fairly steady, progress arising from better computer architectures has been much less consistent.

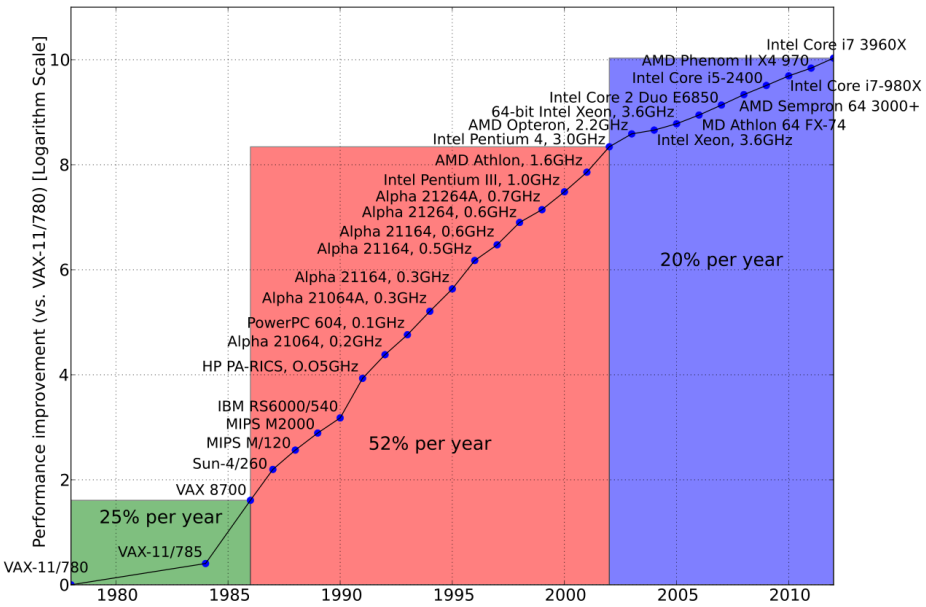


Figure 1.1: Growth in processor performance since the mid-1980s.

The chart, that is presented in Figure 1.1, plots performance relative to the *VAX 11/780* as measured by the *SPECint* benchmarks [SPE12]. Since SPEC (*Standard Performance Evaluation Corporation*) has changed over the years, performance of newer machines is estimated by a scaling factor that relates the performance for different versions of SPEC (*e.g.*, *SPEC92*, *SPEC95*, and *SPEC2000*). Figure 1.1 shows that the combination of architectural and organisational enhancements led to 16 years of sustained growth in performance at an annual rate of over 50 %, a rate that is unprecedented in the computer industry. The effect of this dramatic growth rate in the 20th century has been twofold. First, it has significantly enhanced the capability available to computer users. Second, this dramatic rate of improvement has led to the dominance of microprocessor-based computers across the entire range of the computer design. These innovations led to a renaissance in computer design, which emphasised both architectural innovation and efficient use of technology improvements. This rate of growth has compounded so that by 2002, high-performance microprocessors were

about seven times faster than what would have been obtained by relying solely on technology, including improved circuit design. However, Figure 1.1 also shows that this 16-year renaissance is over. Since 2002, processor performance improvement has dropped to about 20 % per year due to the triple hurdles of maximum power dissipation of air-cooled chips, little parallelism left to exploit efficiently, and almost unchanged memory latency. Indeed, in 2004 *Intel* [INT11] cancelled its high-performance uniprocessor projects and joined *IBM* [IBM12] and *Sun Microsystems* [SUN12] in declaring that the road to higher performance would be via multiple processors per chip rather than via faster uniprocessors. This signals a historic switch from relying solely on ILP (*Instruction-Level Parallelism*), to TLP (*Thread-Level Parallelism*) and DLP (*Data-Level Parallelism*). Whereas the compiler and the hardware conspire to exploit ILP implicitly without the programmer's attention, TLP and DLP are explicitly parallel, requiring the programmer to write parallel code to gain performance. Due to this fact, the introduction of hardware techniques in the design of the system becomes crucial to help the programmer in the task of gaining the required performance.

In the 1960s, the dominant form of computing was on large mainframes-computers costing millions of dollars and stored in computer rooms with multiple operators overseeing their support. Typical applications included business data processing and large-scale scientific computing. The 1970s saw the birth of the minicomputer, a smaller-sized computer initially focused on applications in scientific laboratories, but rapidly branching out with the popularity of multiple users sharing a computer interactively through independent terminals. The 1980s saw the rise of the desktop computer based on microprocessors, in the form of both PC (*Personal Computer*) and workstations. The individually owned desktop system replaced time-sharing and led to the rise of server systems that provided larger-scale services such as reliable, long-term file storage and access, larger memory, and more computing power. The 1990s saw the emergence of the *Internet* and the WWW (*World Wide Web*), and the emergence of high-performance digital consumer electronics, from video games to set-top boxes. The extraordinary popularity of cell phones has been obvious since 2000, with rapid improvements in functions and sales that far exceed those of the PC. These more recent applications use embedded systems, where computers are lodged in other devices and their presence is not immediately obvious. This evolution has set the stage for a dramatic change in how computing is viewed in this new century, which has led to create three different computing markets, each characterised by different applications, requirements, and technologies. Table 1.1 summaries these mainstream classes of computing environments and their important characteristics. Note the wide range in system price for server and embedded systems. For server systems, this range arises from the need for very large-scale multiprocessor systems for high-end transaction processing and WWW server

applications. The total number of embedded systems sold in 2005 is estimated to exceed 3 billion if 8-bit and 16-bit microprocessors are included, instead of 200 million of desktop systems and 10 million of server systems.

Table 1.1: Summary of the three mainstream computing classes and their system characteristics [HP07].

Feature	Desktop Systems	Server Systems	Embedded Systems
Price of system	\$500 – \$5,000	\$5,000 – \$5,000,000	\$10 – \$100,000
Price of microprocessor module (per processor)	\$50 – \$500	\$200 – \$10,000	\$0.01 – \$100
Critical system design issues	Price-performance, graphics performance	Throughput, availability, scalability	Price, power consumption, application-specific performance

The largest computing market in dollar terms as shown in Table 1.1 is the desktop computing. Desktop computing spans from low-end systems that are sold for under \$500 to high-end heavily configured workstations that may be sold for \$5,000. Throughout this range in price and capability, the desktop market tends to be driven to optimise price-performance. As a result, the newest, highest-performance microprocessors and cost-reduced microprocessors often appear first in desktop systems.

The role of server systems grew to provide larger-scale and more reliable file and computing services. The WWW server applications accelerated this trend because of the tremendous growth in the demand and sophistication of WWW-based services. Such servers have become the backbone of large-scale enterprise computing, replacing the traditional mainframe. Unlike desktop systems, server systems are constrained by other characteristics. The first characteristic that is critical is dependability. Failure of a server system, as an international company like *Google* [GOO12] has, is far more catastrophic than failure of a single desktop system, since these server systems must operate seven days a week, 24 hours a day. The second key characteristic of server systems is scalability, because these systems often grow in response to an increasing demand for the services that they support or an increase in functional requirements. Finally, responsiveness to an individual request remains important, but overall efficiency and cost-effectiveness, as determined by how many requests can be handled in a unit time, are key characteristics that are required for most server systems.

Embedded systems, that are the electronic systems in which this Ph.D. thesis is focused on, are the fastest growing portion of the computer market, and

have the widest spread of processing power and cost. They include 8-bit and 16-bit processors that may cost less than a dime, 32-bit microprocessors that execute 100 million instructions per second and cost under $5, and high-end processors for the newest video games or network switches that cost $100 and can execute a billion instructions per second. Although the range of computing power in the embedded computing market is very large, price is a key factor in the design of systems for this space. Performance requirements do exist, of course, but the primary goal is often meeting the performance need at a minimum price, rather than achieving higher performance at a higher price. Embedded systems have different and specific characteristics compared to general-purpose systems. Firstly, embedded systems combine both software and hardware to run a fixed and specific set of applications. These applications differ greatly in their characteristics because they range from multimedia consumer devices to industrial control systems. Due to this fact, embedded systems require different hardware architectures to create an optimum trade-off between performance and cost based on the expected objectives of the target applications. Secondly, unlike general-purpose systems, embedded systems are characterised by restrictive resources and a low-energy budget. Under the constraint of not being mains-connected, the use of an integrated power supplier (*e.g.*, a battery) limits their operation time. Thirdly, in order to be reliable and predictable, embedded systems provide high computational capabilities whereas they satisfy the varied and tight time conflicting constraints that are imposed by the running application. The combination of all these requirements, for a given specific application, makes the design of embedded systems become a big challenge for embedded systems designers.

The *memory wall* [HP07] is a well-known problem in computer systems. It is based on the growing disparity between the rate of improvement in microprocessor speed and the rate of improvement in off-chip memory speed. Figure 1.2 shows as example the clock rate and power for *Intel x86* microprocessors over eight generations and 25 years. As it is possible to see from this example, the *Pentium 4* made a dramatic jump in clock rate and power but less so in performance. The *Pentium Prescott* thermal problems led to the abandonment of the *Pentium 4* line. The *Core 2* line reverts to a simpler pipeline with lower clock rates and multiple processors per chip. This problem becomes even worse in embedded systems, where designers not only need to consider the performance, but also the energy consumption. Several works like [CRL+10], [HP07], and [VM07] demonstrate that the IMO (*Instruction Memory Organisation*) and the DMH (*Data Memory Hierarchy*) take portions of chip area and energy consumption that are not negligible. In fact, both memory architectures now account for up to 40 %–60 % of the total energy budget of an embedded instruction-set processor platform [CRL+10]. M. Verma *et al.* [VM07] did extensive experiments to validate the

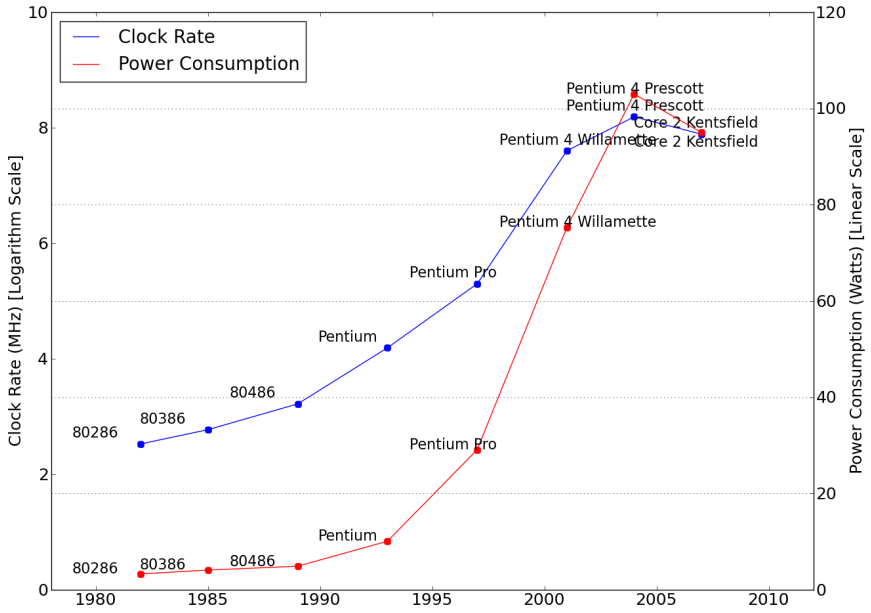
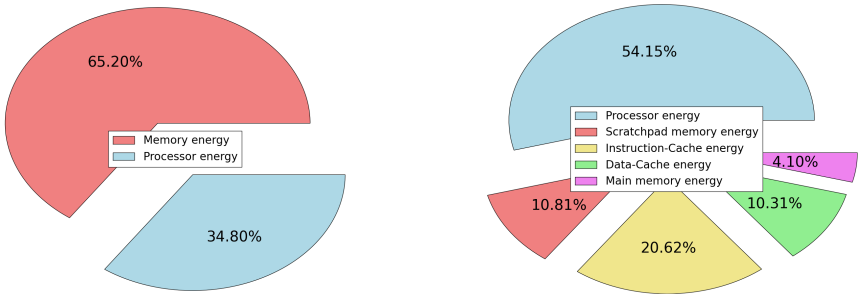


Figure 1.2: Clock rate and power consumption for *Intel x86* microprocessors over eight generations and 25 years.



(a) Uniprocessor *ARM* [Uni12] (b) Multiprocessor *ARM* [BBB+05]

Figure 1.3: Energy distribution in a uniprocessor *ARM* and a multiprocessor *ARM* [VM07].

above observation for embedded systems using *ARM* processors [ARM12]. Figure 1.3 summarises the results of the experiments that were performed in a uniprocessor *ARM* [Uni12] and a multiprocessor *ARM* [BBB+05] based setups. As described in [VM07], more than 150 experiments were conducted to compute the average processor and memory energy consumption values for each one of the two systems. From the Figure 1.3, it is possible to observe that the memory subsystem consumes 65.2 % and 45.9 % of the total energy budget for uniprocessor *ARM* and multiprocessor *ARM* systems, respectively. The main memory for the multiprocessor *ARM* based system is an on-chip SRAM (*Static Random Access Memory*) memory as opposed to off-chip SRAM memory for the uniprocessor system. Therefore, the memory subsystem accounts for a smaller portion of the total energy budget for the multiprocessor system than for the uniprocessor system.

1.1.2. Wireless Sensor Networks

The recent development of high-performance microprocessors and novel sensing materials has stimulated great interest in the development of physical, chemical, or biological sensors combined with integrated circuits. It is not uncommon to place multiple sensors on a single chip, with the integrated circuitry of the chip controlling all these sensors.

The applications of these smart sensors are extensive, and describing all the uses of these devices is probably not possible. However, a few example application domains will help the reader of this document understand the importance of these smart sensors and the interest on them. On the one hand, it is possible to see the military interest based on the following examples. First, networks of smart sensors could be deployed in combat scenarios to track troop movements. Second, smart sensors placed on small robots could conduct land mine detection. Third, smart sensors could detect the use of biological or chemical weapons and, via network communication, report their presence in time to protect troops. On the other hand, civilian applications of smart sensors are also numerous and varied. One example is pollution detection along beaches, with smart sensors distributed along the shoreline and using wireless communication to relay information to a base station for further processing. Other example could be the distribution of smart sensors throughout the exhaust system of an automobile to detect levels of emissions and efficiently reduce pollution.

This list of applications only scratches the surface of proposed and potential applications of novel smart sensors. However, one thing in common with the aforementioned applications is that using a wireless interface to these devices is superior to a wired connection, even in cases where a wired connection may

be possible. A distributed WSN (*Wireless Sensor Network*) is formed by several scattered nodes, from hundreds to thousands, in a sensor field. Each node contains both processing and communication elements and has as main functionality event-oriented environment monitoring. Collected data from the environment are sent to the base station in order to be processed. Thanks to the great node density of this kind of networks, the collaboration among them allows the creation of a high quality and failure resistant environment monitoring system [SCL+04, BCDV09]. Due to the fact that in this case the smart sensors are included in wireless networks, these smarts sensors can be in this case applied in a wider variety of applications, such as:

- Continuous patient monitoring.
- In-building people localisation for efficient energy control.
- Aircraft fatigue breakage supervision.
- Dangerous and harmful agents' detection in great traffic density areas.
- Tornado evolution analysis.
- Forest fire, earthquake, or flooding detection.
- Remote terrains monitoring.
- Environmental danger detection.
- Metropolitan area traffic study for routes planning.
- Free parking spot control.
- Home, mall, public buildings, and some others facilities surveillance and security.
- Military applications for detecting, locating, or tracking enemy movement.
- Potential terrorist attacks alert.
- Vineyard management.
- Interactive museums or toys.
- Domotics.

As these applications show, embedded electronic devices will be a part of our future daily life. Due to both the limited energy budget and the computational capabilities, the smart sensors that are used for biological implants present research challenges such as the need for having at the same time a bio-compatible, fault-tolerant, energy-efficient, and scalable design. The biomedical application domain perfectly points out the low-energy budget of these systems as the main research challenge. Embedded systems are characterised by restrictive resources and a low-energy budget. However, biomedical wireless sensor nodes add additional constraints to the energy budget of the embedded system. The heat that is dissipated from the energy consumption of the embedded system has to be carefully controlled. For instance, depending on where the sensor is placed in the body, the dissipated heat cannot be allowable, because small increases of temperature can already damage the human tissues [SGW01]. Due to this fact, power awareness plays a

vital role in the life of a WSN. But it is important not only to manage existing resources correctly in order to avoid wasting them and reserve some of them for critical situations, but also to regenerate or harvest consumed energy as much as possible to make the network live longer. The secret to reduce both energy consumption and dissipation so much lies in power aware design of every layer of the system.

Due to the fact that this Ph.D. thesis is focused on embedded systems, real-life embedded applications of the specific subdomain of wireless sensor nodes are used as benchmarks to show, analyse, and corroborate the benefits and disadvantages of each one of the concepts in which this Ph.D. thesis is based on. The selected benchmarks are described in Appendix C.

1.2. Problem Formulation

A typical embedded system architecture consists of an instruction-set processor core, a reconfigurable hardware unit, an instruction cache, a data cache, an on-chip SPM (*Scratchpad Memory*), an on-chip DRAM (*Dynamic Random Access Memory*), and an off-chip main memory [GLW05]. As shown in Figure 1.4, the computations are partitioned into different computational units, while the data are assigned to different storage components. Unlike the design of the memory hierarchy of general-purpose systems, which is focused on the performance of the system, the design of the memory hierarchy of embedded systems has more diverse objectives where area, performance, bandwidth communication, and energy consumption are included with similar weights in the system design.

The problem of the *memory wall* in embedded systems can be seen in a realistic illustrative example based on a configurable and low-power mixed signal SoC (*System-on-Chip*) for portable ECG (*Electrocardiogram*) monitoring applications [KYS+11]. This example, which can be understood as a representative sensor node application, is based on a mixed-signal ECG SoC that is capable of implementing configurable functionality with low-power consumption for portable ECG monitoring applications, as it can be seen in Figure 1.5. With the increasing use of ambulatory monitoring systems, not only continuous signal collection and low-power consumption, but also smartness with robust operation under the presence of signal artifacts is required. In particular, robust operation necessitates advanced functionalities such as motion artifact removal and accurate R peak detection leading to increased computation [YKT+10]. However, previous solutions using general-purpose processors have limited functionality [CBW+09] or cannot achieve very low-power consumption [WMK+08]. This drives the integration of the

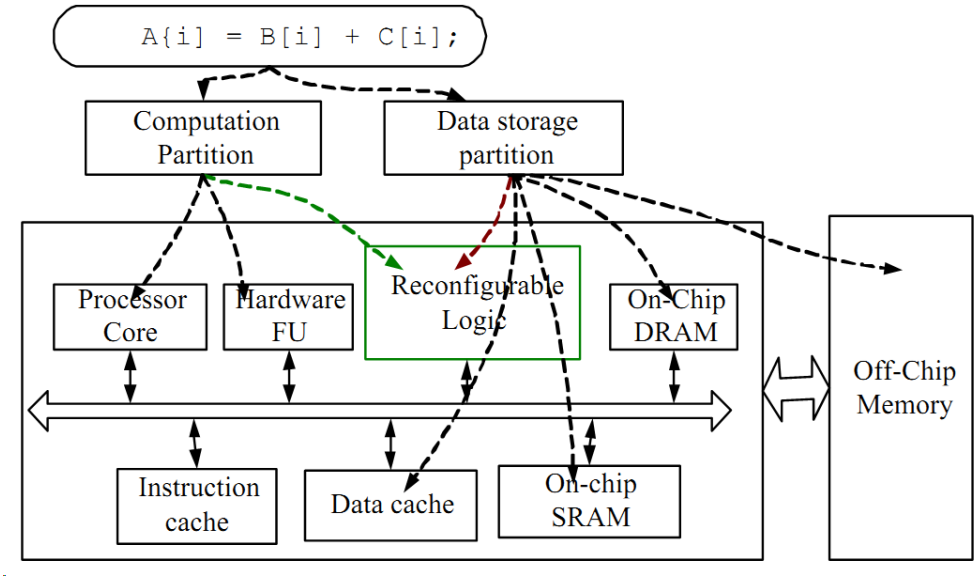


Figure 1.4: A typical embedded system architecture [GLW05].

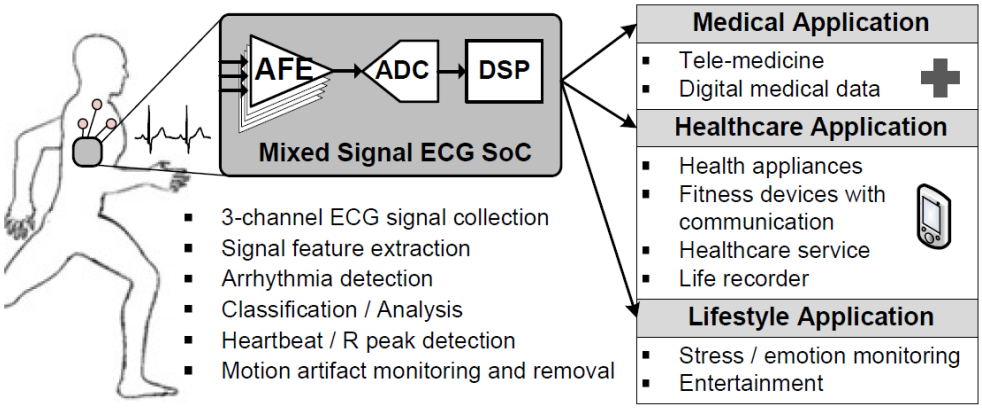


Figure 1.5: Mixed-signal ECG SoC and typical applications [KYS+11].

proposed mixed signal SoC combining a low-power analog front-end with a fully optimised and configurable DSP (*Digital Signal Processor*) back-end. The target of this design is to enable configurability for different applications, ranging from simple heart rate calculation towards more complex medical diagnostics under ambulatory conditions, with low-power consumption and high accuracy. The block diagram of the digital signal processor back-end is shown in Figure 1.6.

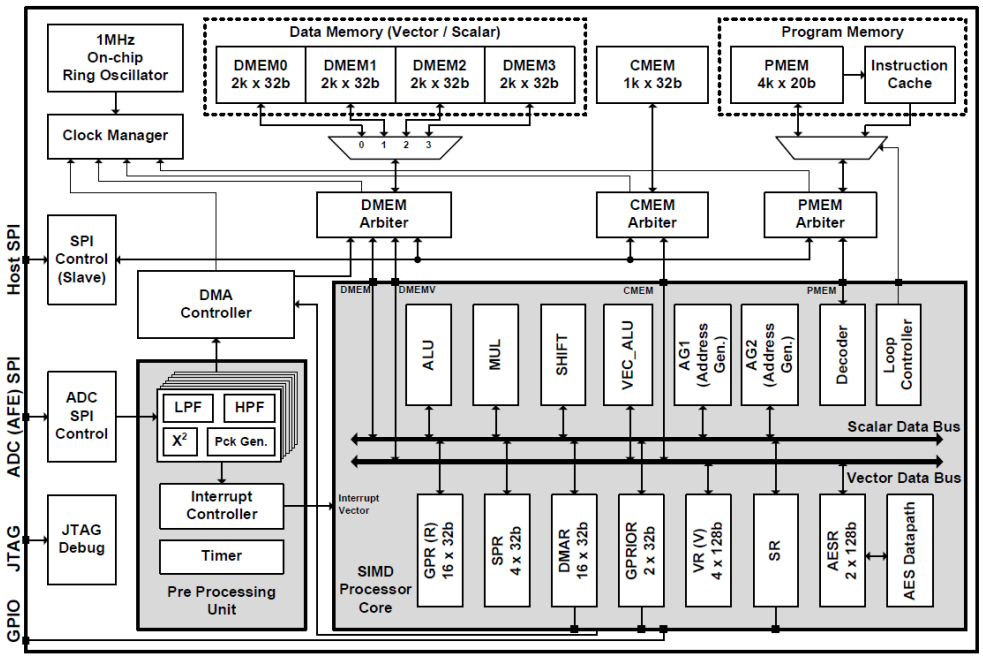


Figure 1.6: Block diagram of the digital signal processor back-end [KYS+11].

The power breakdown of this real-life embedded system, that is based on a configurable and low-power mixed signal SoC for portable ECG monitoring applications, can be seen in Figure 1.7. In this Figure, the heartbeat detection algorithm that is described in Section C.6 is being executed in the real-life embedded system. From this Figure 1.7, where the components of the processor core are grouped, it is possible to see that both instruction memory organisation and data memory hierarchy strongly dominate the energy consumption of the embedded system. This case study proves why further optimising instruction memory organisations from the energy consumption point of view will remain an extremely important trend in the future. This will be the focus of this Ph.D. thesis.

From the literature study of this Ph.D. thesis, that is extensively presented in Chapter 2, it is possible to recognise that future research on instruction

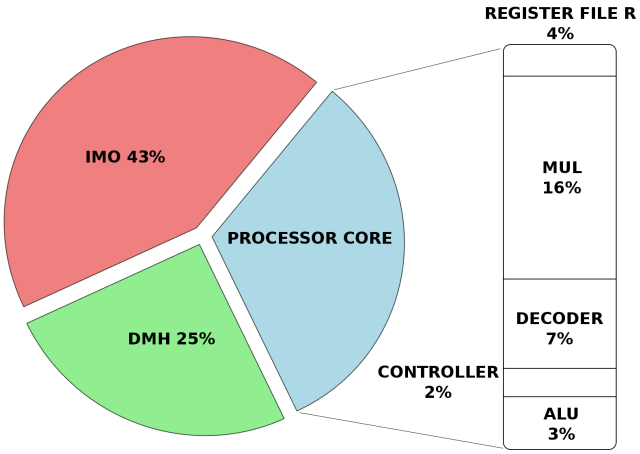


Figure 1.7: Power breakdown of a biomedical wireless sensor node running a heartbeat detection algorithm [YKH+09].

memory organisations will be focused on the development of enhancements and optimisations that increase the exploitation of parallelism in architectures not only to improve the performance, but also to become more energy efficient. In order to achieve this purpose, future designers of instruction memory organisations will have to keep in mind that an increase in the parallelism of the system can be directly related to an increase in performance but not necessarily in energy efficiency. For instance, heavily partitioned instruction memory organisations are better in energy efficiency than centralised instruction memory organisations shown by the results that are exposed in the analysis of the efficient partitioning of the L1 instruction cache (Section 2.3). Nevertheless, due to the overhead that is introduced in address decoding and control logic, heavily partitioned instruction memory organisations may have a worse delay and hence longer clock cycle. Therefore, it can be concluded that the more centralised is the instruction memory organisation, the more reduction in the delay can be achieved. This fact is true, only if the memory is internally optimised. However, these architectures can be worse in the energy consumption per memory access. The design/compilation complexity will be also taken into account in future designs of the instruction memory organisation. As can be seen in Section 2.3.4, distributed loop buffer architectures with incompatible loop-nest organisation are the best in performance due to the management of incompatible loop-nests. However, the design complexity of this kind of architectures is high. In this case, due to this drawback, researchers will use co-design to make an optimal trade-off between multiple and distributed loop buffer architectures. This example provides a nice illustration of the trade-off that exists between performance and design/compilation complexity. The concern on the design of

the instruction memory organisation is appearing slowly due to the fact that electronic systems are starting to be characterised by restrictive resources and low-energy budgets. Therefore, it will be crucial to introduce any enhancement in the instruction memory organisation to allow not only to decrease the total energy consumption, but also to have a better distribution of the energy budget throughout the system. Despite that the *Core 2* architecture [INT11] has the first architectural enhancement that is starting to be applied, the number of implementations on commercial devices of the enhancements that are described in Chapter 2 will be increased considerably in the coming future.

1.3. Overview of the State-of-the-art

During the last years, the research community has analysed and implemented enhancements in the data memory hierarchy and communication network that have led to improvements on both of these subjects. However, the amount of research carried out in the area of the instruction memory organisation that is related to energy optimisation is relatively limited in the existing literature. Some issues like cache hardware improvements, instruction decoding, and instruction scheduling have been managed to some extent (see Chapter 2). Nevertheless, these advances have not yet solved the energy bottleneck sufficiently well.

Due to the fact that hardware and software are tightly-coupled in embedded systems, optimisations can be done either in the separate domains or preferably combined across domains. In this Section, the analysis of the whole set of techniques that are used to improve the instruction memory organisation is split in Section 1.3.1 and Section 1.3.2. Section 1.3.3 presents the research issues that still remain open based on the state-of-the-art overview that is presented in Section 1.3.1 and Section 1.3.2.

1.3.1. Summary of Hardware Optimisations

Embedded systems designers customise the instruction memory organisation based on the conclusions that were obtained from previous analysis of specific applications. Several works like [HP07], [VM07], and [CRL+10] have demonstrated that larger memories mean more power consumption for the system. Although, the emphasis on low power is frequently driven by the use of batteries, the need to use less expensive packaging (*e.g.*, plastic versus ceramic) and the absence of a fan for cooling also limit the total power consumption of the embedded system.

J. Villarreal *et al.* [VLCV01] show that 77% of the total execution time of an application is spent in loops of 32 instructions or less. This fact demonstrates that in applications of signal and image processing, a significant amount of the total execution time is spent in small program segments. If these small program segments can be stored in smaller memory banks (*e.g.*, in the form of a LB (*Loop Buffer*)), the dynamic energy consumption of the system can be reduced significantly. Figure 1.8 shows that accesses to a small memory have lower energy consumption than to a large memory. This observation is the base of the loop buffer concept, which is a scheme to reduce the dynamic energy consumption in the instruction memory organisation. Besides, memory banking is identified as an effective method to reduce the leakage energy consumption in memories [KKK02]. Apart from the possibility of using multiple low-power operating modes, the use of memory banks reduces the effective capacitance as compared to a single monolithic memory.

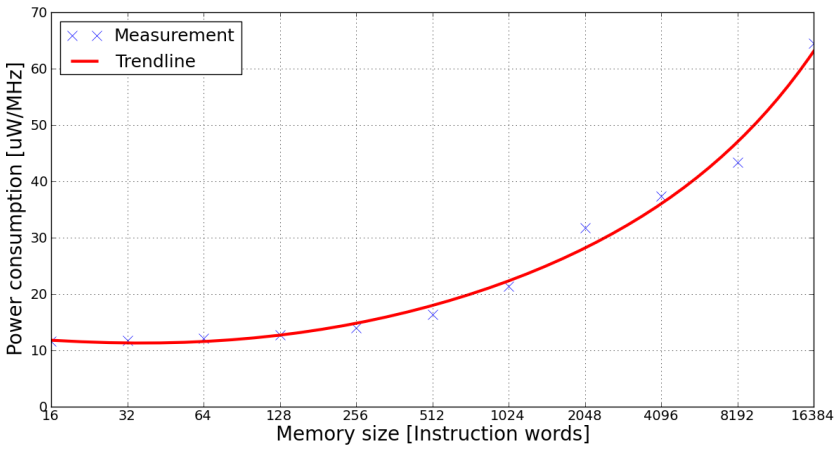


Figure 1.8: Power consumption per access in 16-bit word SRAM-based memories designed by *Virage Logic Corporation* tools [VIR12] using TSMC CMOS 90nm process.

During the last decade, researchers have demonstrated that, the energy consumption of the instruction memory organisation is not negligible as shown in Section 1.2, and loop buffering is an architectural enhancement that is introduced in the instruction memory organisation that allows, not only to decrease the total energy consumption, but also to have a better distribution of the energy budget throughout the embedded system. The CELB (*Central Loop Buffer Architecture for Single Processor Organisation*) represents the most traditional use of the loop buffer concept. R. S. Bajwa *et al.* [BHK+97] proposed an architectural enhancement that could switch off the fetch and decode logic if loops could be identified, fetched, and decoded only once. The instructions of the loop were decoded and stored locally, from where the

instructions were executed. The energy savings came from the reduction of memory accesses as well as the lesser use of the decode logic. In order to avoid any performance degradation with the introduction in the embedded system of a loop buffer, L. H. Lee *et al.* [LMA99] implemented a small instruction buffer based on the definition, detection, and utilisation of special branch instructions. This architectural enhancement had neither an address tag store nor valid bit associated with each loop cache entry. J. Kin *et al.* [KGMS97] evaluated the *Filter Cache*. This enhancement was an unusually small first-level cache that sacrificed a portion of performance in order to save energy. The PM (*Program Memory*) was only required when a miss occurred in the *Filter Cache*, otherwise it remained in standby mode. In addition, K. Vivekanandarajah *et al.* [VSB04] presented an architectural enhancement that detected the opportunity to use the *Filter Cache*, and enabled or disabled it dynamically. Also, W. Tang *et al.* [TGN02] introduced a DFC (*Decoder Filter Cache*) in the instruction memory organisation in order to reduce the use of the instruction fetch and decode logic by providing directly decoded instructions to the processor. Figure 1.9 shows the generic architecture of this set of instruction memory organisations. This architecture neither has partitioning in the loop buffer architecture nor in the program memory, and its connections depend on a single centralised component. Therefore, efficient parallelism in the execution of an application cannot be achieved in this kind of architectures due to the lack of hardware resources.

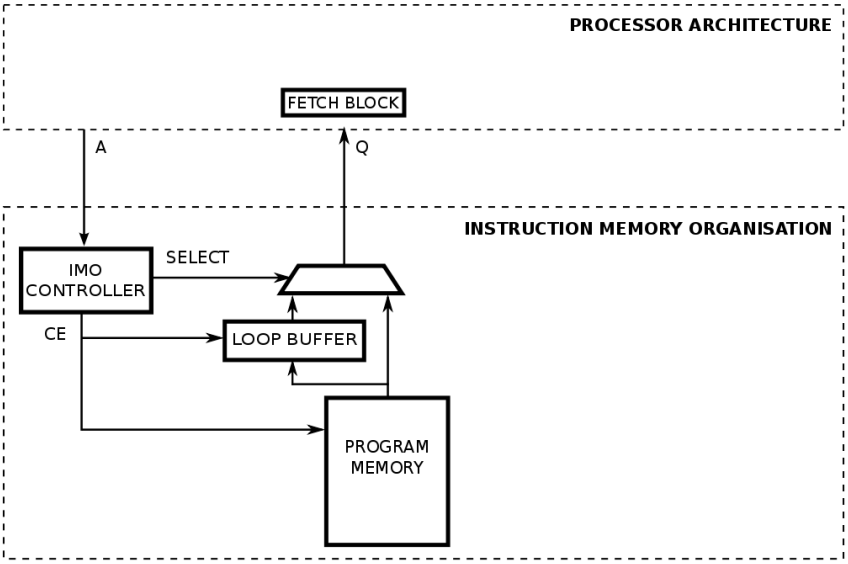


Figure 1.9: Instruction memory organisation with a central loop buffer architecture for single processor organisation.

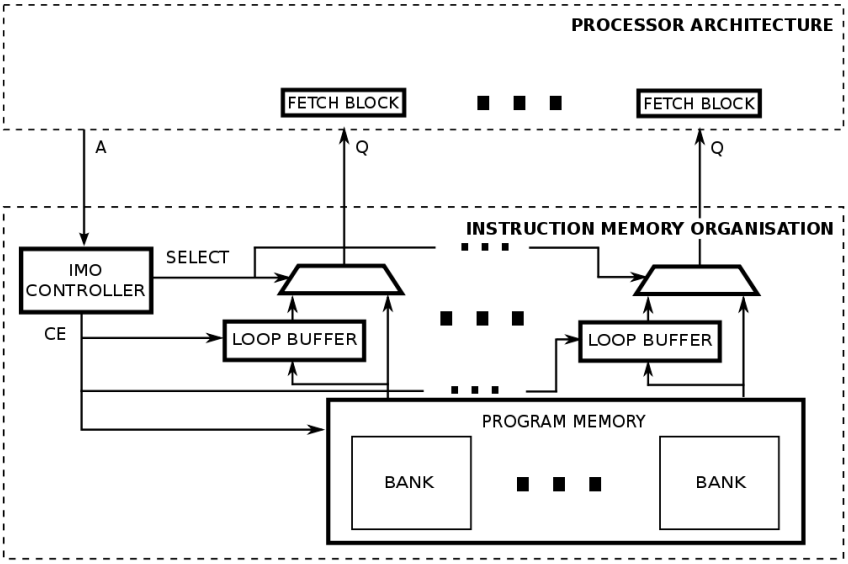


Figure 1.10: Instruction memory organisation with a clustered loop buffer architecture with shared loop-nest organisation.

Parallelism is a well-known solution in order to increase performance efficiency in an embedded system. Since loops form the most important part of a program [VLCV01], techniques like loop fusion and other loop transformations are applied to exploit the parallelism (boosting ILP) within loops on single-threaded architectures. However, the centralised resources and global communication of single-threaded architectures make CELB architectures less energy efficient, when loop transformation techniques are applied to them in order to exploit parallelism within loops. The CLLB (*Clustered Loop Buffer Architecture with Shared Loop-Nest Organisation*) mitigates these bottlenecks. H. Zhong *et al.* [ZFMS05] presented a distributed control-path architecture for DVLIW (*Distributed Very Long Instruction Word*) processors that overcame the scalability problem of control-paths in VLIW (*Very Long Instruction Word*) architectures. The architectural enhancement that was presented in this work was to distribute the fetch and decode logic in the same way that the register file is distributed in a multi-cluster data-path. Also, H. Zhong *et al.* [ZLM07] proposed a multi-core architecture that extended traditional multi-core systems in two ways. First, this architecture provided a dual-mode scalar operand network to enable efficient inter-core communication without using the memory. Second, this architecture organised the cores for execution in a way that created a trade-off between communication latency and flexibility. D. Black-Schaffer *et al.* [BSBD+08] analysed a set of architectures for efficient delivery of VLIW instructions. A baseline cache implementation was compared to a variety of organisations,

where the evaluation included the cost of the memory accesses and the wires that were necessary to distribute the instruction bits. Figure 1.10 shows the generic architecture of a CLLB architecture, which inner connections are controlled by one single component. In this architecture, the controller is more complex, because it controls the partitions that exist in the loop buffer architecture and the program memory. This set of architectures also includes all the enhancements that come from the introduction of low-power operating modes, as well as from the research that was done in power management of banked memories [BMP00, FEL01, LK04].

An efficient parallelism exploitation is not yet fully achievable with the CLLB architecture. Using CLLB architectures, loops with different threads of control have to be merged into a single loop with a single thread of control. This code transformation is performed using techniques like loop transformations (*e.g.*, loop fusion). However, not all loops of an application can be efficiently exploited in this manner. In the case of incompatible loops, the parallelism cannot be efficiently exploited because they require multiple loop controllers, which results in loss of energy and performance. Therefore a need exists for a multi-threaded platform that could support execution of multiple incompatible loops with minimal hardware overhead. That is achievable with the DLB (*Distributed Loop Buffer Architecture with Incompatible Loop-Nest Organisation*). M. Jayapala *et al.* [JBB+02] proposed a low-energy clustered instruction memory organisation for VLIW processors. In the proposed architecture, a simple profile based algorithm was used in order to perform an optimal synthesis of the clusters for a given application. P. Raghavan *et al.* [RLJ+06] presented a multi-thread distributed instruction memory organisation that supported execution of multiple incompatible loops in parallel. In the proposed architecture, each loop buffer had its own local controller, which was responsible for indexing and regulating accesses to its loop buffer. J. I. Gomez *et al.* [GMV+04] presented a code transformation that optimised the memory bandwidth, based on the combination of loops with non-conformable headers. With this technique of code transformation, the compiler could then better exploit the available bandwidth and increase the performance of the system. Figure 1.11 shows the generic architecture of this recent representative loop buffer architecture. As shown, the inner connections of this instruction memory organisation are managed by a logic of controllers that is distributed across the architecture. In this instruction memory organisation, partition exists in both the loop buffer architecture and the program memory. The controller of this memory is even more complex than the controllers of the previous ones, because it also controls the execution of each loop in the corresponding loop buffer to allow the parallel execution of loops with different iterators.

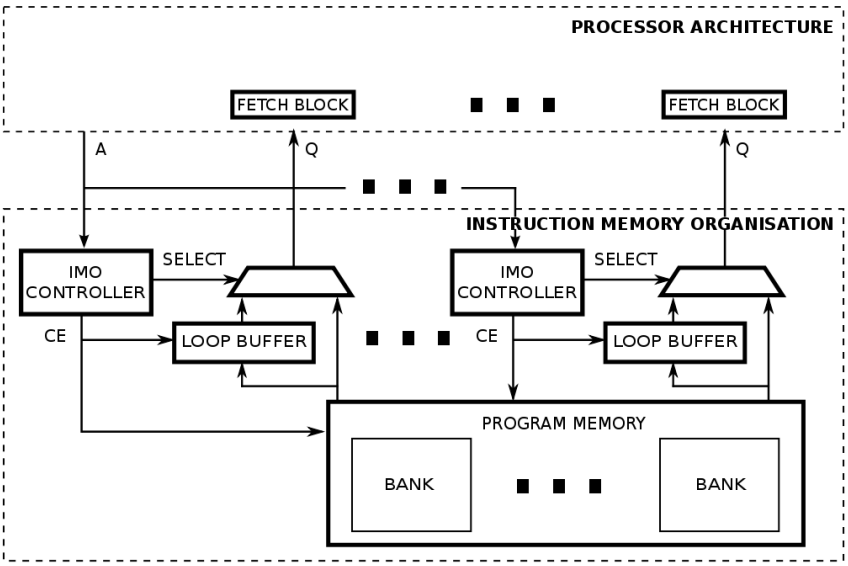


Figure 1.11: Instruction memory organisation with a distributed loop buffer architecture with incompatible loop-nest organisation.

Summarising, the three generic architectures where all loop based architectures can fall in are:

- CELB (*Central Loop Buffer Architecture for Single Processor Organisation*).
- CLLB (*Clustered Loop Buffer Architecture with Shared Loop-Nest Organisation*).
- DLB (*Distributed Loop Buffer Architecture with Incompatible Loop-Nest Organisation*).

It should be noted that in the industry, the loop buffer concept is starting to be applied. The *Core 2* architecture of *Intel* [INT11] can use the decoder queue as a 64-bytes loop buffer, which is organised as four lines of 16 bytes each. The four 16-bytes blocks do not have to be consecutive. The architecture uses a hardware loop detection mechanism, called LSD (*Loop Stream Detector*), that detects small loops inside of the IQ (*Instruction Queue*). Once a loop is detected, instructions for subsequent loop iterations are streamed from the IQ without any external fetching, until a misprediction on the loop branch is detected.

Section 2.3 discusses extensively hardware optimisations of the instruction memory organisation.

1.3.2. Summary of Software Optimisations

The application domain characteristics and the architecture design can be combined. However, the work of the embedded designer is not finished yet because the application has to be mapped to the architecture. The performance, the energy consumption, and the behavioural predictability are directly related to the software that is running on the embedded system. Since embedded applications are quite loop intensive, tackling power consumption of loop execution is a source of major benefits.

In terms of profiling for code optimisation, D. Marculescu [Mar00] addressed the problem of energy optimisation by using a compiler-assisted technique for code annotation that adaptively selected at run-time the optimal number of instructions to be fetched or executed in parallel. Based on experimental results, this approach was up to 23 % better than clock throttling and was as efficient as voltage scaling. K. Inoue *et al.* [IMM02] proposed an approach to detect and remove unnecessary tag-checks at run-time. Using execution footprints that were recorded previously in a BTB (*Branch Target Buffer*), it was possible to omit the tag-checks for all instructions contained in a fetched block.

In terms of source code transformations, T. Ishihara *et al.* [IY00] proposed a technique which merged frequently executed sequences of object codes into a set of single instructions, which reduced the number of accesses to the main memory dramatically. This technique had an energy reduction of more than 65 % in the best case compared to an instruction memory organisation without this enhancement. N. D. Liveris *et al.* [LZSG02] proposed a flow that iteratively applied source code transformations to improve performance in the instruction memory organisation. The procedure was driven by a set of analytical equations that predicted the number of misses based on the parameters that were related to the application code and the structure of the instruction memory organisation.

In terms of code mapping, K. Pettis *et al.* [PH90] presented several algorithms of code positioning. The first algorithm, that was built on top of the linker, positioned the code based on procedure invocations. The second algorithm, that was built on top of an optimiser package, positioned code based on the basic blocks that existed within procedures. As everything was done at compile time, no penalty on execution time was observed. From the results of this work it was possible to see that more benefits can be reached from the basic block level rather than from the procedure level. T. Vander Aa *et al.* [AJB+03] presented a framework to optimise the energy consumption of the instruction memory organisation in VLIW processors. This work showed that code transformations (*e.g.*, loop peeling, factorisation, loop re-rolling,

and loop splitting) were needed not only to increase the loop buffer coverage of the application, but also to optimise the energy consumption.

Section 2.4 discusses extensively software optimisations of the instruction memory organisation.

1.3.3. Problem Statement

The embedded systems designer faces a complex task. First, the designer has to determine which attributes are important for a new system, then to design a system to maximise performance while staying within cost, power, and availability constraints. This task has many aspects, including instruction set design, functional organisation, logic design, and implementation. The implementation may encompass integrated circuit design, packaging, power, and cooling. Optimising the design requires familiarity with a very wide range of technologies, from compilers and operating systems to logic design and packaging. The state-of-the-art overview that is summarised in Section 1.3.1 and Section 1.3.2, and which is extensively analysed in Chapter 2, provides enough familiarity with the wide range of specific problems that the research community has solved. However, it can be concluded that the existing techniques, that are focused on the reduction of the energy consumption of the instruction memory organisation, can be classified based on the following main trends:

- The use of small memories in the form of memory banks for the implementation of the L0 instruction cache (*i.e.*, the loop buffer memory).

- The improvement of the efficiency in the partitioning of the L1 instruction cache.

- The addition of enhancements in the compiler to improve the mapping of the application and make an efficient use of the architectural enhancements that can be introduced in the instruction memory organisation.

After providing an up-to-date picture of the current status of instruction memory organisations, and giving to the reader a first grasp on the fundamental characteristics and design constraints of various types of instruction memory organisations, it can be concluded that the following problems remain open in order to achieve the objective of providing an instruction memory organisation with a low-cost energy per task:

- The use of post-layout simulations to evaluate the energy impact of loop buffer architectures in an experimental evaluation with a strict method in order to have an accurate estimation of parasitics and switching activity.
- Because of the ever-increasing complexity and size of embedded systems, a high-level energy estimation tool is required during the design process of an embedded system to increase the simulation speed and the energy savings. Previous works have showed sophisticated energy modelling methods to precisely estimate the energy consumption at system-level [BLRC05], [ANMD07], [KAA+07], and [LKY+06]. However, these methods lack a design space exploration, from the energy consumption point of view, of the different architectural options that are used to implement the instruction memory organisation. This high-level energy estimation tool can allow embedded systems designers to perform a high-level trade-off analysis of different types of loop buffer schemes for instruction memory organisations, which can be used not only to show which scheme is more suitable for applications with certain behaviour, but also to present the correct process design that an embedded systems designer has to follow in order to have an efficient implementation of a loop buffer scheme for a certain application.
- As it is possible to see after comparing different types of loop buffer schemes for instruction memory organisations, embedded systems designers face a trade-off between the energy budget of the system and the performance that is required by the application that is running on the system. As shown in Section 1.3.1 and Section 1.3.2, conventional loop buffer architectures (*i.e.*, CELB architectures and CLLB architectures) are not good enough in terms of energy efficiency due to the high overhead that these loop buffer architectures show. Due to this fact, DLB architectures appear as a promising option to improve the energy efficiency of instruction memory organisations. There is a need of the proposition and analysis of different options to implement efficient DLB architectures for a given application.

1.4. Contributions of this Ph.D. Thesis

This Ph.D. thesis is focused on the analysis and implementation of hardware techniques for low-energy instruction memory organisations in embedded systems. The main contributions that this Ph.D. thesis has for the research community are:

- The systematic study of existing low-energy optimisation techniques that are used in instruction memory organisations, outlining their comparative advantages, drawbacks, and trade-offs.

- The use of post-layout simulations to evaluate the energy impact of different loop buffer architectures in an experimental evaluation with a strict method in order to have an accurate estimation of parasitics and switching activity.
- The development of a high-level energy estimation tool that, for a given application and compiler, allows the exploration not only of architectural and compiler configurations, but also of code transformations that are related to the instruction memory organisation.
- The evaluation of different loop buffer schemes for certain embedded applications, guiding the embedded systems designer to make the correct decision in the trade-offs that exist between energy budget, required performance, and area cost of the embedded system.
- An implementation of a run-time loop buffer architecture that optimises both the dynamic and leakage energy consumption of the instruction memory organisation.
- The proposal and analysis of non-overlapping and complementary implementation options for distinct partitions of the design space that is related to DLB architectures.

In terms of scientific publications, this Ph.D. thesis has generated the following articles in international journals:

ARJ+b13 Artes, A., R. Fasthuber, J. L. Ayala, P. Raghavan, and F. Catthoor, "Design Space Exploration of Loop Buffer Schemes in Embedded Systems", Special Issue of Journal of Systems Architecture on Design Space Exploration of Embedded Systems: Elsevier Amsterdam, 2013.

ARJ+a13 Artes, A., R. Fasthuber, J. L. Ayala, P. Raghavan, and F. Catthoor, "Design Space Exploration of Distributed Loop Buffer Architectures with Incompatible Loop-Nest Organisations in Embedded Systems", Journal of Signal Processing Systems: Springer New York, 2013.

AJJFa12 Artes, A., J. L. Ayala, J. Huisken, and F. Catthoor, "Survey of Low-Energy Techniques for Instruction Memory Organisations in Embedded Systems", Journal of Signal Processing Systems: Springer New York, 2012.

A.JFb12 Artes, A., J. L. Ayala, and F. Catthoor, "Power Impact of Loop Buffer Schemes for Biomedical Wireless Sensor Nodes", Journal of MDPI Sensors: MDPI AG, 2012.

, this Ph.D. thesis has generated the following articles in international peer-reviewed conferences:

A.JFa12 Artes, A., J. L. Ayala, and F. Catthoor, "IMOSIM: Exploration Tool for Instruction Memory Organisations based on Accurate Cycle-Level Energy Modelling", IEEE International Conference on Electronics, Circuits, and Systems (ICECS), 2012.

AJV+a11 Artes, A., J. L. Ayala, V. A. Sathanur, J. Huisken, and F. Catthoor, "Run-time Self-tuning Banked Loop Buffer Architecture for Power Optimization of Dynamic Workload Applications", IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SOC), 2011.

A. Aa10 Artes, A., "Power Consumption on Loop Buffer based Instruction Memory Organizations", STW.ICT Conference on Research in Information and Communication Technology, 2010.

AFM+a09 Artes, A., F. Duarte, M. Ashouei, J. Huisken, J. L. Ayala, D. Atienza, and F. Catthoor, "Energy Efficiency using Loop Buffer based Instruction Memory Organization", IEEE International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA), 2009.

, and this Ph.D. thesis has contributed in the following articles in international journals and peer-reviewed conferences:

HSN+a13 Kim, H., S. Kim, N. V. Helleputte, A. Artes, M. Konijnenburg, J. Huisken, C. V. Hoof, and R. F. Yazicioglu, "A Configurable and Low-Power Mixed Signal SoC for Portable ECG Monitoring Applications", Journal of IEEE Transactions on Biomedical Circuits and Systems: IEEE Computer Society, 2013.

MMJ+a12 Komalan, M., M. Hartmann, J. I. Gomez, C. Tenllado, A. Artes, and F. Catthoor, "System Level Exploration of Resistive-RAM (ReRAM) based Hybrid Instruction Memory Organization", International Memory Architecture and Organization Workshop (MeAOW), 2012.

HRS+a11 Kim, H., R. Firat, S. Kim, V. N. Helleputte, A. Artes, M. Konijnenburg, J. Huisken, J. Penders, and V. C. Hoof, "A Configurable and Low-Power Mixed Signal SoC for Portable ECG Monitoring Applications", IEEE International Symposium on VLSI Technology and Circuits, 2011.

1.5. Structure of this Ph.D. Thesis

The rest of the document of this Ph.D. thesis is organised as follows:

- Chapter 2 provides a synthesis on the low-energy techniques that are used in instruction memory organisations, outlining their comparative advantages, drawbacks, and trade-offs.

- Chapter 3 proposes a high-level energy estimation tool that, for a given application and compiler, allows the exploration not only of architectural and compiler configurations, but also of code transformations that are related to the instruction memory organisation.

- Chapter 4 presents a high-level analysis of promising loop buffer schemes that exist in embedded systems, and proposes a run-time loop buffer architecture that optimises both the dynamic and the leakage energy consumption of the instruction memory organisation.

- Chapter 5 shows how the loop buffer concept is applied in real-life embedded applications that are widely used in biomedical wireless sensor nodes, to show which scheme of loop buffer is more suitable for applications with certain behaviour.

- Chapter 6 proposes and analyses non-overlapping and complementary implementation options for distinct partitions of the design space that is related to DLB architectures.

- Chapter 7 synthesises the conclusions derived from the research that is presented in this Ph.D. thesis, and the contributions to the state-of-the-art for the development of loop buffer architectures for instruction memory organisation. This Chapter also includes a future work summary.

- Appendix A includes a Spanish summary of this dissertation, in compliance with the regulations of the *Universidad Complutense de Madrid*.

- Appendix B presents the tool that has been used in the design of the processor architectures that are presented throughout this Ph.D. thesis.

- Appendix C includes a description of each one of the benchmarks that are used along this Ph.D. thesis.

Figure 1.12 provides to the reader not only an overview of the Chapters that form this Ph.D. thesis, but also how these Chapters interact among them. For instance, it is possible to see how the high-level analyses, that are presented in Chapter 4, Chapter 5, and Chapter 6, are based on the high-level energy estimation and exploration tool that is presented in Chapter 3.

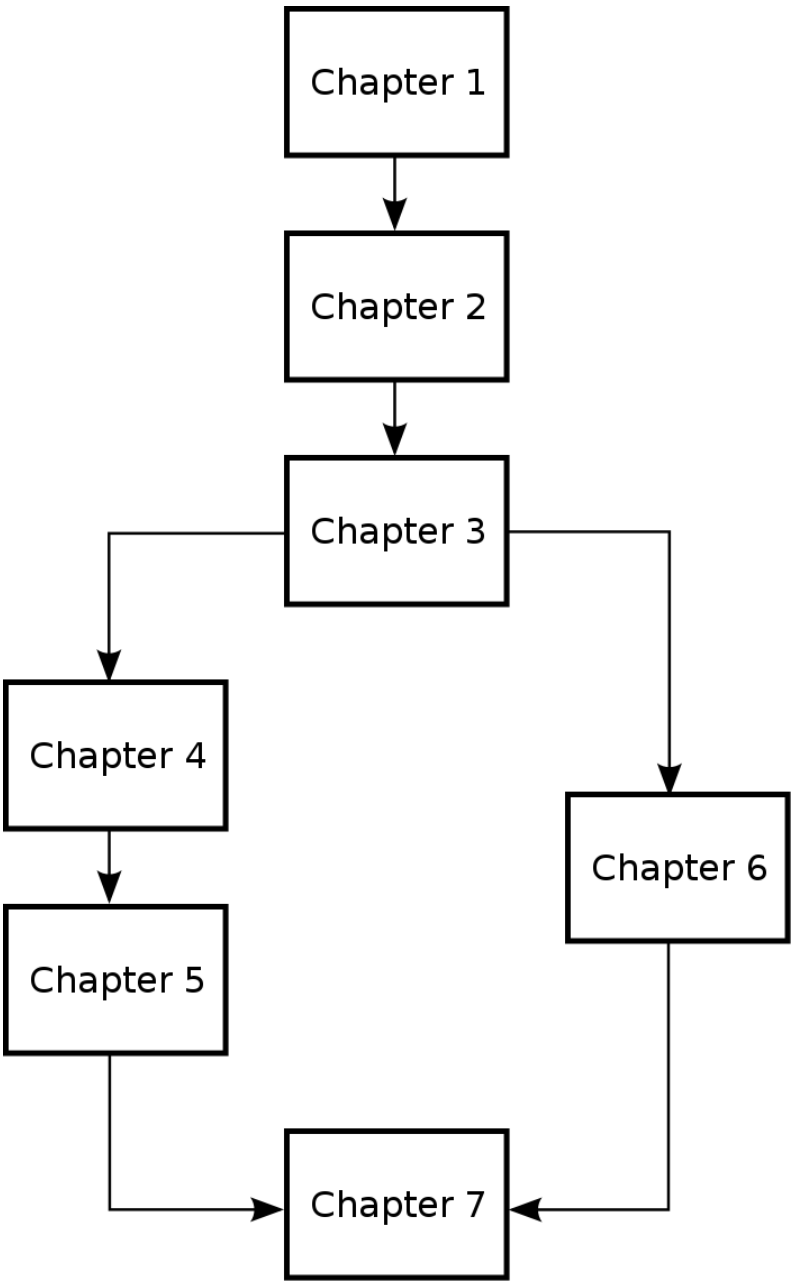


Figure 1.12: Overview of the Chapters that form this Ph.D. thesis.

In the next chapter...

the reader will get a synthesis on the low-energy techniques that are used in instruction memory organisations for embedded systems, outlining their comparative advantages, drawbacks, and trade-offs.

Chapter 2

Survey of Low-Energy Techniques for Instruction Memory Organisations in Embedded Systems

"Never walk on the traveled path because it only leads where others have been."

— Alexander Graham Bell.

The work that is presented in this Chapter provides a synthesis on the low-energy techniques that are used in instruction memory organisations, outlining their comparative advantages, drawbacks, and trade-offs. Apart from giving to the reader a first grasp on the fundamental characteristics and design constraints of various types of instruction memory organisations, the architectural classification that is presented in this Chapter has the advantage of clearly exhibiting lesser explored techniques, and hence providing hints for future research on the instructions memory organisations that are used in embedded systems.

2.1. Introduction

Embedded systems have different and specific characteristics compared to general-purpose systems. Firstly, embedded systems combine both software and hardware to run a fixed and specific set of applications. These applications differ greatly in their characteristics because they range from multimedia consumer devices to industrial control systems. Due to this fact, embedded systems require different hardware architectures to create an optimum trade-off between performance and cost based on the expected objectives of

the target applications. Secondly, unlike general-purpose systems, embedded systems are characterised by restrictive resources and a low-energy budget. Under the constraint of not being mains-connected, the use of an integrated power supplier (*e.g.*, a battery) limits their operation time. Thirdly, embedded systems usually work as reactive systems, because embedded systems are connected to the physical world through sensors, and they have to react to every external *stimuli* that is received by them. Therefore, in order to be reliable and predictable, embedded systems provide high computational capabilities whereas they satisfy the varied and tight time constraints that are imposed by the running application.

The wireless sensor nodes applications form an important subdomain of the overall embedded application domain as is described in Section 1.1.2. In this subdomain, embedded systems contain multiple smarts sensors on a single chip. These smart sensors, in which sensing materials are combined with the integrated circuitry that controls them, require capabilities to communicate themselves with external base stations via wireless interfaces. Due to both the limited energy budget and the computational capabilities, the smart sensors that are used for biological implants present research challenges such as the need for having at the same time a bio-compatible, fault-tolerant, energy-efficient, and scalable design. Although the biomedical application domain has a lot of interest nowadays, it perfectly reflects the main research challenges:

- **Real-time performance requirements.** A real-time performance requirement is when a segment of the application has to obey an absolute maximum execution time. In the nodes of a WSN (*Wireless Sensor Network*), the time to process each data frame is limited, since the processor must process the next frame shortly. For more information about this aspect, Q. Zhou *et al.* [ZXL07] provides a detailed real-time performance analysis for WSNs.
- **Low-energy budget.** Wireless sensor nodes add additional constraints to the low-energy budget of embedded systems. The heat that is dissipated from the energy consumption of the system has to be carefully controlled. For instance, depending on where the sensor is placed in the human body, the dissipated heat cannot be allowable [SGW01]. Besides, the absence of wires to supply a constant source of energy causes that the use of an energy harvesting source [CC08] or an integrated energy supplier [AG09] limits the operation time of these devices.
- **Efficient management of the bandwidth communication.** As wireless sensor nodes are dealing with a small and finite quantity of energy, and wireless communication is vital and very expensive in terms of energy consumption [WLLP01], it is not surprising that little energy remains for computation (*i.e.*, local processing). This limited computation is seen as a counter argument of the use of a LB (*Loop*

Buffer) in the IMO (*Instruction Memory Organisation*), because of the fact that the complexity of the on-sensor application code is increased in order to manage the architecture of the loop buffers. However, the reality is that due to the too expensive wireless communication that is needed to transmit the data at the originally required sample rate, local processing becomes important in order to send through wireless communication exclusively the relevant data of the information that is sensed by the wireless sensor node. Therefore, wireless communication should be used only when it is really needed, which allows to reduce the overall energy budget. Hence, the computational load increases and so is the contribution of the corresponding IMO in the total energy budget. That motivates again the need for effective ways to keep the overhead of the IMO negligible. It is necessary to stress that at the end, the design engineer is who sets the point in the trade-off curve between the energy that is consumed by the wireless communication and the computation that is performed inside of the wireless sensor node.

The combination of these requirements makes wireless sensor nodes significantly different from other types of embedded systems. As a result, their design space exploration reveals a wide range of specific problems that the research community has to solve. Figure 2.1 summarises the main design choices that an embedded designer can take. From this Figure, it is possible to see that designs that are based on an ASIC (*Application-Specific Integrated Circuit*) are the best choice in terms of performance and energy efficiency, DSP (*Digital Signal Processor*) offers flexibility as well as delivers performance but is not as energy efficient as an ASIC, and ASIP (*Application-Specific Instruction-Set Processor*) offers flexibility apart from the possibility to reuse for a given application the energy characteristics of the ASIC design. Because wireless sensor nodes require embedded systems with low-energy consumption, certain flexibility to be reused for other domains in order to reduce the production and chip cost per device, and a low design effort, most of these embedded systems are designed using the ASIP design style from the first stages of the design [SWKS01]. It should be noted that the design effort to map code on an ASIP is high, but an efficient ASIP design compensates this drawback through performance and energy efficiency.

In order to optimise an ASIP, embedded designers must evaluate the requirements and constraints of the application in order to reduce the energy consumption per task. This reduction has to be done fulfilling the required real-time constraints of the system. In this way, an optimum trade-off is created between the performance and the energy consumption of the system. It is necessary to stress that reducing the peak power consumption of the system is not the same as reducing the energy per task, because this last concept also takes into account the application workload.

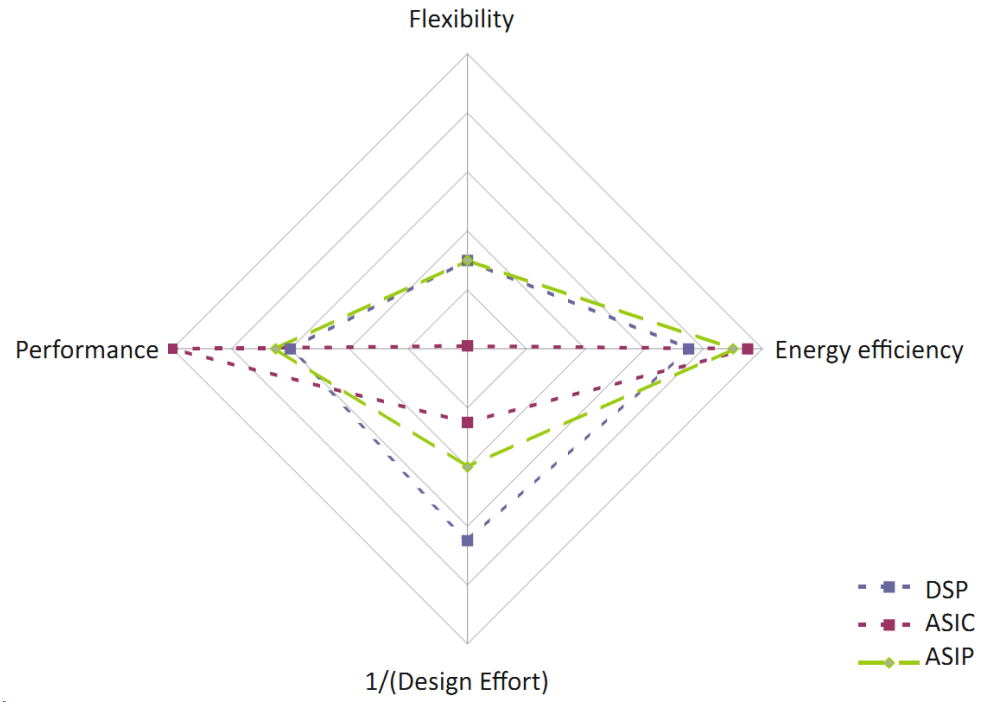


Figure 2.1: Design styles based on design metrics [CRL+10].

2.2. Approaches

A sample block diagram of a typical embedded system is shown in Figure 2.2. As it is possible to see from this Figure, embedded systems encapsulate all the devices such as processor architecture, memory storage, interface, and control logic in a single package or board to perform only a set of specific application tasks. Due to this fact, it is possible to generalise that a typical embedded system architecture consists of an instruction-set processor core, a reconfigurable hardware unit, an instruction cache, a data cache, an on-chip SPM (*Scratchpad Memory*), an on-chip DRAM (*Dynamic Random Access Memory*), and an off-chip main memory [GLW05]. As shown in Figure 2.3, the computations are partitioned into different computational units, while the data are assigned to different storage components. Unlike the design of the memory organisation of general-purpose systems, which is focused on the performance of the system, the design of the memory organisation of embedded systems has more diverse objectives where area, performance, bandwidth communication, and energy consumption are included with similar weights in the system design.

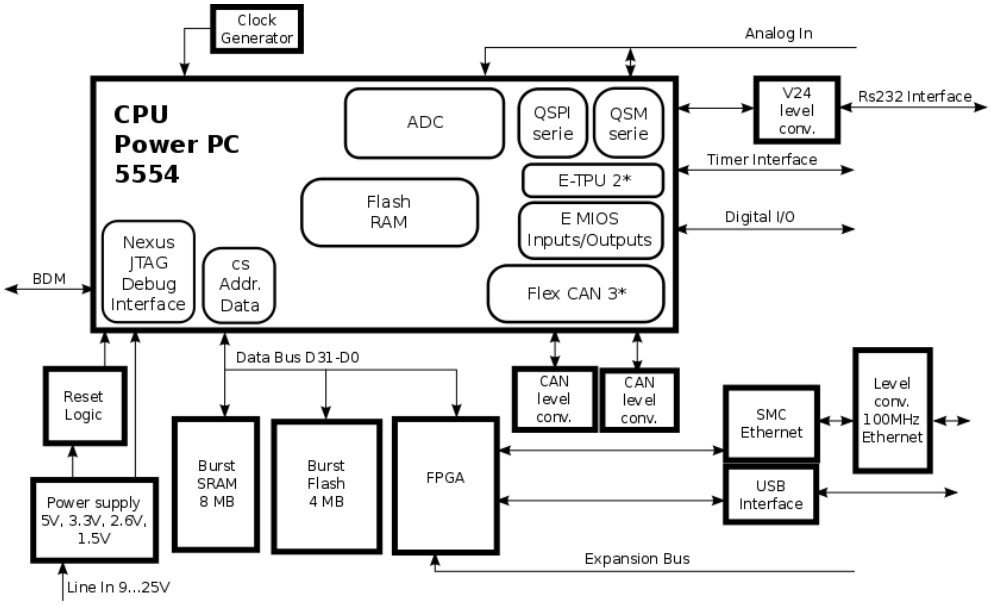


Figure 2.2: Sample block diagram of a typical embedded system.

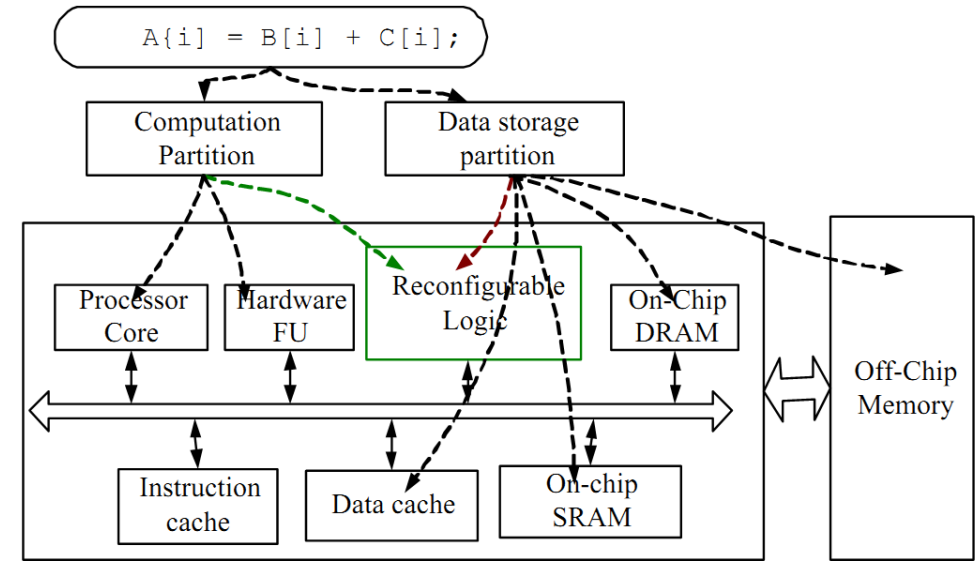


Figure 2.3: A typical embedded system architecture [GLW05].

The memory bottleneck in a modern desktop system is a widely known problem. This problem, which was introduced in Section 1.1.1, is characterised by the fact that the memory speed cannot follow the processor speed. Over the past 30 years, the speed of computer systems grew at a phenomenal rate of 50 %–100 % per year, whereas during the same period, the speed of a typical DRAM grew at a modest rate of about 7 % per year. Nowadays, the extremely fast microprocessors spend a large number of cycles idle, waiting for the requested data to arrive from the slow memories. This fact leads to the problem, also known as the *memory wall problem*, in which the performance of the entire system is not governed by the speed of the processor but by the speed of the memory. Figure 2.4 plots the processor performance projections against the historical performance improvement in time to access the main memory. Research works like [HP07] and [VM07] provide a more detail information of this problem.

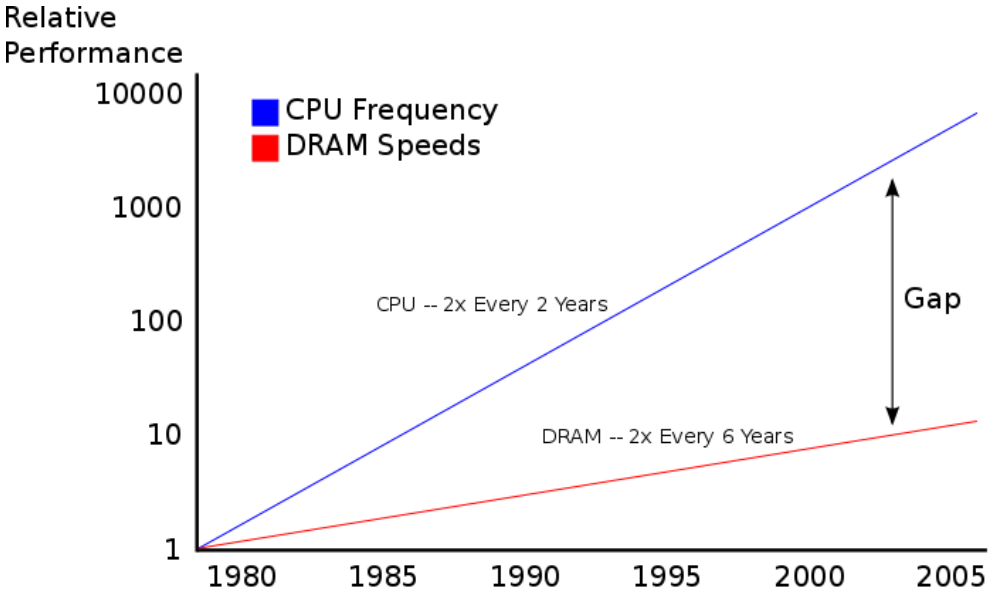


Figure 2.4: Processor performance projections against the historical performance improvement in time to access the main memory [SUN12].

The well-known problem of the *memory wall* becomes even worse in embedded systems, where designers not only need to consider the performance, but also the energy consumption. Several works like [CRL+10], [HP07], and [VM07] demonstrate that the IMO and the DMH (*Data Memory Hierarchy*) take portions of chip area and energy consumption that are not negligible. In fact, both memory architectures now account for up to 40 %–60 % of the total energy budget of an embedded instruction-set processor platform [CRL+10]. As example, Figure 2.5 presents a power breakdown of an embedded

instruction-set processor platform where an advanced encryption standard algorithm is running on an ASIP [TSH+10]. This Figure shows the components of the processor core grouped. From Figure 2.5, it is possible to see that both IMO and DMH strongly dominate the energy consumption of the embedded system. During the last years, the research community has analysed and implemented enhancements in the DMH and communication network which has led to improvements on both of these subjects. However, the amount of research carried out in the area of the IMOs that is related to energy optimisations is relatively limited in the existing literature. Some issues like cache hardware improvements, instruction decoding, and instruction scheduling have been managed to some extent (see Section 2.3 and Section 2.4). Nevertheless, these advances have not solved yet the energy bottleneck sufficiently well. The case study that is presented in Figure 2.5 provides the proof why further optimising IMOs from the energy consumption point of view will remain an extremely important trend in the future.

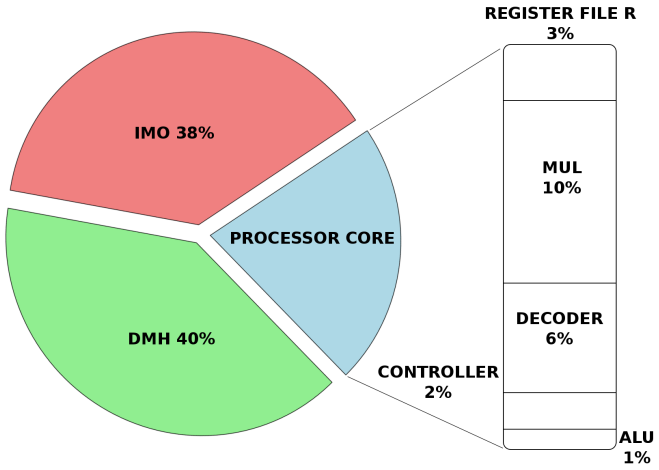


Figure 2.5: Power breakdown of a biomedical wireless sensor node running an advanced encryption standard algorithm [TSH+10].

Due to the fact that hardware and software are tightly-coupled in embedded systems, optimisations can be done either in the separate domains or preferably combined across domains. In order to ease the analysis of the techniques that are used to improve the IMO, this Chapter categorises them into the traditional approaches: hardware and software. The first approach deals with architectural aspects. Embedded designers customise the memory hierarchy based on conclusions that were obtained from previous analysis of specific applications. Section 2.3 discusses extensively hardware optimisations of the IMO. The second approach deals with the software aspect. After being able to combine the application domain characteristics and the architecture

design, the application has to be mapped on the architecture. For this reason, embedded designers analyse and optimise the application intensively, such as partitioning data and/or instructions into different types of storage, or optimising data and/or program layouts to reduce the amount of cache misses. The key software techniques are explained in detail in Section 2.4. Although, works like Z. Ge *et al.* [GLW05] claim that it is necessary to combine hardware and software in order to explore the design space more thoroughly and make a better adaptation between these two aspects, this Chapter presents and analyses these approaches separately. The idea proposed in [GLW05] is indeed feasible for an application specific approach, but it is not feasible for instruction-set processors which have to serve for a wide range of applications.

2.3. Hardware Optimisations

Numerous thesis have provided options to solve the problem of the energy consumption in IMOs. In this Section, enhancements that deal with the architectural aspect of the system are addressed. These enhancements look at two aspects: the energy consumption of the memory architecture and the access time delay to the memory space. Therefore, the use of the metric *energy per task* in the design is crucial, because both aspects play a role in this metric.

In order to analyse the energy consumption of the IMO, it is interesting to examine the components that form this organisation. One well-known and frequently used component is the instruction cache. Cache memories are usually based on a SRAM (*Static Random Access Memory*) for performance reasons. DRAMs are typically slower, due to the refresh, and therefore less desirable when they are used on battery powered systems. The major work, that has been done to improve the energy consumption in cache memories, is related to the reduction of the accesses to the main memory for general-purpose computing. Nevertheless, when looking into the embedded systems domain, there is not much work that is related to the management of alternative components that form the architecture of the IMO.

In order to understand how the access time delay of a memory is related to the architecture of the IMO, the design metric *miss rate* has to be taken into account. Misses appear when accesses are performed to memory locations not present locally. Due to the long distance and therefore high capacitance of the (mostly off-chip) buses and the large storage of off-chip memories, accesses to these memories consume a lot of time and energy. Any enhancement in the architecture leading to reduce *miss rate* directly reduces not only energy consumption, but also access time. Therefore, the performance and the energy consumption of the system both profit from architectural enhancements.

The following Sections describe the state-of-the-art of the architectural enhancements that have been developed for the optimisation of the architecture of the IMO.

2.3.1. Direct-Mapped Cache Memories

Direct-mapped caches are popular in embedded microprocessor architectures, since set-associative caches are typically only targeting performance improvement, but at an unacceptable energy efficiency loss. Direct-mapped caches with just one cache line per set consume less energy per access than a same sized set-associative cache, because set-associative caches have to access multiple cache ways simultaneously.

N. P. Jouppi [Jou90] analysed three hardware techniques to improve direct-mapped cache performance: miss caching, victim caching, and stream buffer. Firstly, in the miss caching technique a small fully-associative cache (named miss cache) was introduced between the first-level cache and the access port to the second-level cache. When a miss occurred, data were fetched and stored not only in the direct-mapped cache, but also in the introduced miss cache. If the miss occurred in the direct-mapped cache but a match was found in the miss cache, then the direct-mapped cache was reloaded in the next cycle from the miss cache. Secondly, in order to avoid the appearance of the same data in both caches, victim caching used a different strategy for the replacement in the miss cache, allowing to load the miss cache only with items thrown out from the direct-mapped cache. Thirdly, the stream buffer technique reduced capacity and compulsory misses by prefetching cache lines starting at the address where the miss had occurred. The lines after the required line were placed in the buffer architecture and not in the main cache, avoiding polluting the cache with data that may never be needed. The work that is presented in [Jou90] showed that depending on the technique that was used, the miss rate could be reduced by a factor of two or three. Indeed, the higher the percentage of misses due to conflicts, the more effective was the miss cache in eliminating them. This fact was more beneficial to IMOs than DMHs because instruction conflicts tended to be widely spaced, whereas data conflicts were quite closely spaced. However, stream buffer was a better architectural enhancement in DMHs, because more data conflict misses were removed in them than in IMOs.

K. Inoue *et al.* [IMM02] proposed an approach to detect and remove unnecessary tag-checks at run-time. The proposed architecture, HBTC (*History-Based Tag-Comparison*), used execution footprints that were recorded previously in a BTB (*Branch Target Buffer*) to omit the tag-checks for all instructions contained in a fetched block. The execution footprints

contained all the cache lines that were referenced without any cache miss. Therefore, the cache dynamically utilised the contents of the BTB to determine whether the target instructions were inside of the instruction cache without performing tag-checks. The HBTC implementation scheme is shown in Figure 2.6. This technique required neither a large timing-area overhead in the cache nor additional compiler support. However, controlling the execution footprints might lead to some performance degradation. Besides, the invalidation of the execution footprint that was caused by cache misses or BTB replacements produced one stall cycle due to the access conflicts in the BTB. However, this penalty could be hidden if it was smaller than the cache-miss penalty. In any case, simulations showed that this architecture could reduce the total amount of tag-checks up to 95 %, and saved up to 17 % of the energy consumed by the direct-mapped instruction cache, with less than 0.2 % performance degradation.

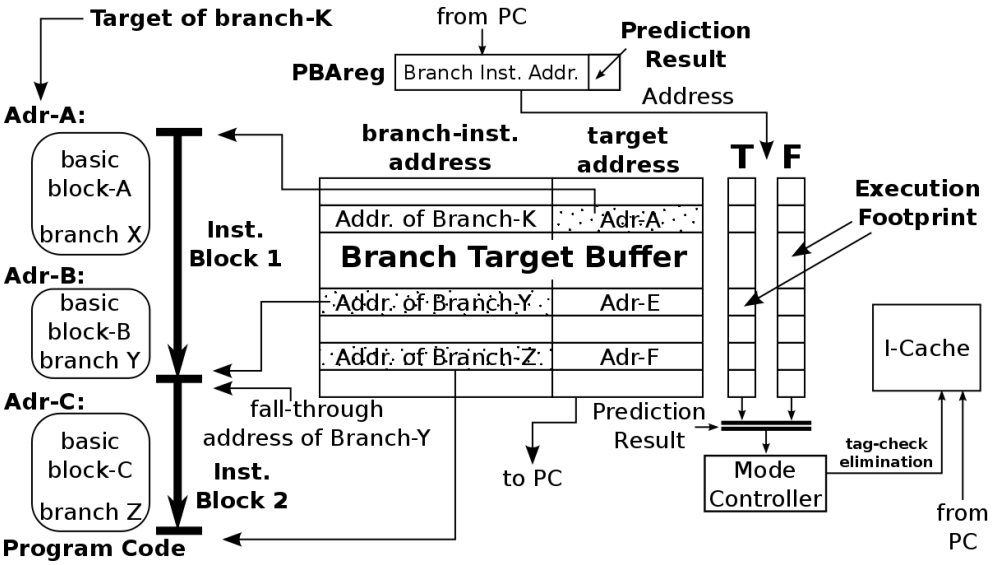


Figure 2.6: The HBTC implementation scheme [IMM02].

2.3.2. Central Loop Buffer Architectures for Single Processor Organisation

Loop buffering is an effective scheme to reduce energy consumption in the IMO. J. Villarreal *et al.* [VLCV01] showed that 77 % of the total execution time of an application was spent in loops of 32 instructions or less. This fact demonstrated that, in applications of signal and image processing, a significant amount of execution time was spent in small program segments. If these small program segments could be stored in smaller memory banks (*e.g.*, in

the form of a LB (*Loop Buffer*)), the energy consumption of the system that was related to circuit logic of the instruction fetch could be reduced significantly. Figure 2.7 shows that accesses in a small memory have lower energy consumption than in a large memory. This observation is the basis of the concept of loop buffering. Like any enhancement technique that is focused on data or instruction locality, loop buffering allows the decrease in the energy consumption through the reduction of the miss rate in the first-level cache.

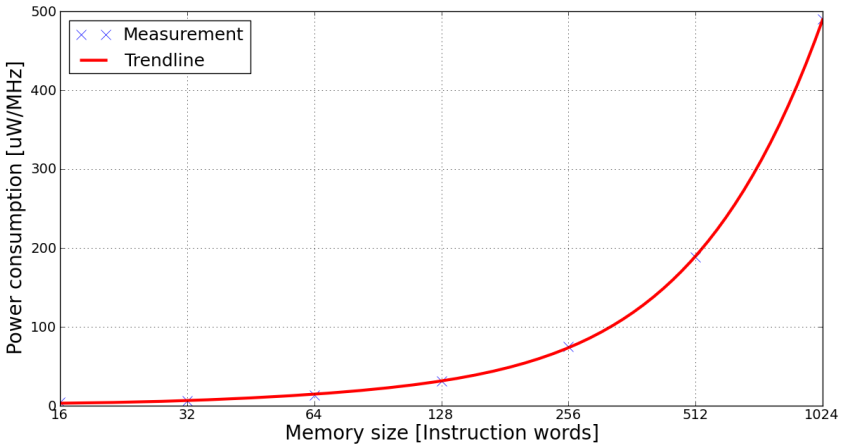


Figure 2.7: Power consumption per access in 16-bit word FF-based memories designed by *Virage Logic Corporation* tools [VIR12] using TSMC CMOS 90nm process.

The CELB (*Central Loop Buffer Architecture for Single Processor Organisation*) is the commercial version of the loop buffering concept, because it is the only one that is starting to be applied. For instance, the *Core 2* architecture of *Intel* [INT11] uses the decoder queue as a 64-bytes loop buffer, which is organised as four lines of 16 bytes each. The four 16-bytes blocks do not have to be consecutive. The architecture uses a hardware loop detection mechanism, called LSD (*Loop Stream Detector*), that detects small loops inside of the IQ (*Instruction Queue*). Once a loop is detected, instructions for subsequent loop iterations are streamed from the IQ without any external fetching, until a mis-prediction on the loop branch is detected.

J. Kin *et al.* [KGMS97] implemented and evaluated the Filter Cache. This cache was an unusually small first-level cache that reduced energy dissipation relative to the traditional cache architecture, albeit at the expense of decreased hit ratio. The idea behind this design was that the decrease in energy consumption compensated for the loss in performance. In this design, the PM (*Program Memory*) was only required when a miss occurred in the Filter

Cache, otherwise it remained in standby mode. The evaluation of the Filter Cache showed that it had faster access time, due to the fact that it was smaller than a first-level cache. Besides, associativity tended to have a strong impact on energy consumption, because it increased the amount of data that was read out of cache arrays. Simulations showed that the energy that was dissipated decreased 58 % in comparison with a baseline system without Filter Cache. However, the execution time was increased 21 %.

Due to the relevance of the work that was presented by J. Kin *et al.* [KGMS97], several works appeared aiming for the improvement of this architecture. K. Vivekanandarajah *et al.* [VSB04] presented an architectural enhancement that detected the opportunity to use the Filter Cache, and enabled or disabled it dynamically. W. Tang *et al.* [TGN02] designed a DFC (*Decoder Filter Cache*) to provide directly decoded instructions to the processor, reducing the use of the instruction fetch and decode logic thereby saving energy. N. Bellas *et al.* [BHPS99] proposed a scheme where the compiler reduced the possibility of a miss in the loop cache by selecting statically which instructions had to be placed in it. With this strategy, the compiler alleviated the negative effects that Filter Caches had on performance and energy. Profiling information was used to select the basic blocks that would be placed in the loop cache based on their frequency of execution. Once the compiler split the basics blocks, the selected blocks were placed before the non-selected ones in the memory address space trying to minimise mapping conflicts in the loop cache. For that purpose, the selected blocks that were in the same nest do not map to the same location. The drawback of this scheme was that when the application had small sections of sequential code, procedural abstraction, or lack of very deeply nested loops (three or more levels), this scheme did not have any benefits. The proposed solution to this lack of benefits was to select a function and place its most important basic blocks permanently in the loop cache.

Chapter 4 and Chapter 5 present high-level analyses in which CELB architectures are evaluated to show the trade-offs that these loop buffer architectures have between energy budget, required performance, and area cost of the embedded system.

2.3.3. Clustered Loop Buffer Architectures with Shared Loop-Nest Organisation

One of the main drawbacks of the architectural enhancements that are described in Section 2.3.2 is the increase in execution time that they suffer in comparison to a baseline system without any enhancement. Due to the fact that parallelism is a well-known solution to increase performance efficiency,

loop transformation techniques have been applied to exploit ILP (*Instruction-Level Parallelism*) within loops on single-threaded architectures to make VLIW (*Very Long Instruction Word*) architectures very effective achieving high performance. Centralised resources and global communication make these architectures less energy efficient. In order to reduce these bottlenecks, several solutions that use multiple loop buffers have been proposed in literature. The general idea is to distribute small instruction memories, which are low energy, over groups of functional units to minimise the overall energy consumption of the system. Figure 2.8 presents some examples of VLIW organisations. In a VLIW organisation, the centralised register file is broken down into several smaller register files. Each one of them supplies operands to a subset of the functional units. The energy consumption in the interconnections is reduced by localising the data transfers. Despite these architectures bring great benefits in relation to the architectures that are described in Section 2.3.2, they are only applied in research environments. References [BSBD+08], [ZFMS05], and [ZLM07] are examples of the work done in this field.

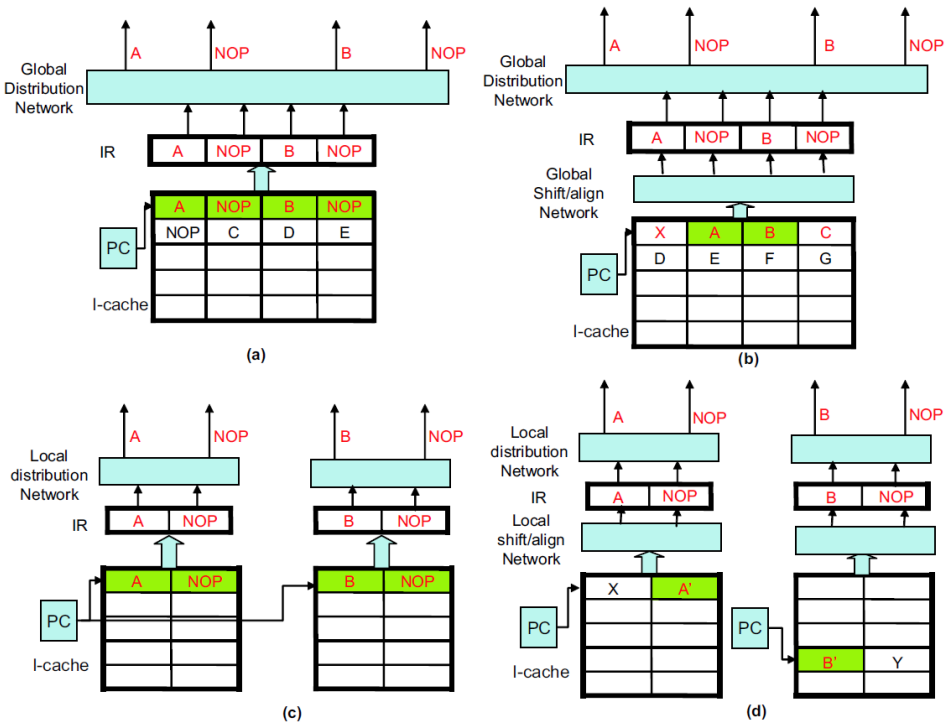


Figure 2.8: VLIW organisations: (a) Centralised instruction cache, uncompressed encoding; (b) Centralised instruction cache, compressed encoding; (c) Distributed instruction cache, centralised PC, uncompressed encoding; (d) Distributed instruction cache, distributed PC, compressed encoding [ZFMS05].

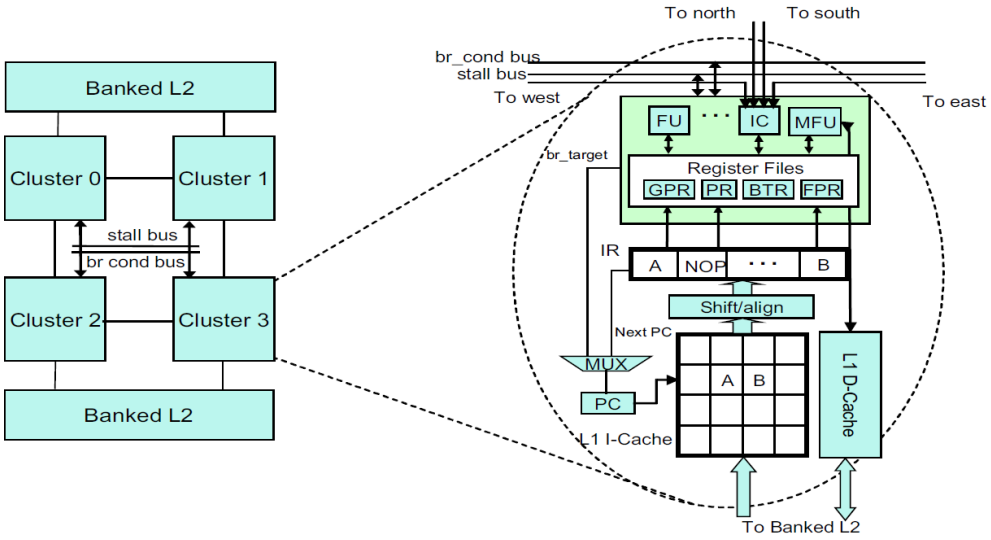


Figure 2.9: DVLIW architecture overview [ZFMS05].

H. Zhong *et al.* [ZFMS05] presented a distributed control-path architecture for DVLIW (*Distributed Very Long Instruction Word*) processors, that overcame the scalability problem of VLIW control-paths. The main idea was to distribute the fetch and decode logic in the same way that the register file was distributed in a multi-cluster data-path. Inter-cluster data communication (*e.g.*, stall signals, broadcasting branch conditions) were handled via connectors between neighbouring clusters. Figure 2.9 shows an example based on four clusters. The latency of inter-cluster communication was exposed to the compiler, which scheduled explicit inter-cluster move operations to transfer data between clusters. All operations in a logical instruction were fetched and executed at the same cycle in different clusters. If any cluster incurred a cache miss, all clusters must stall. A software-controlled mechanism was proposed to handle idle clusters. The idea was to allow the compiler to insert operations that explicitly placed a cluster in sleep mode, and to wake up a cluster to resume execution. The goal of the compiler was to localise communication of data values within a cluster as much as possible, while distributing work across clusters, to take advantage of instruction and loop level parallelism. This led to a small performance overhead due to two aspects. First, extra operations were needed to be inserted for every branch on every cluster, which led to increase the size of the code. Second, there was a small performance overhead in explicitly managing the instruction streams. One of the main contributions of the DVLIW processor was the combination of architecture and compiler support to distribute the program counter, as well as the support of a flexible instruction compression technology in order to provide efficient usage of the IMO.

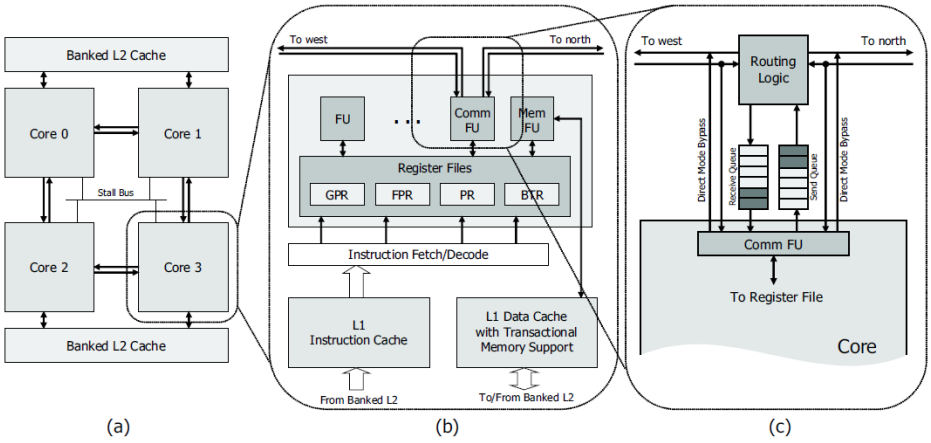


Figure 2.10: Block diagram of the *Voltron* architecture: (a) 4-core system connected in a mesh topology; (b) Data-path for a single core; (c) Details of the inter-core communication unit [ZLM07].

H. Zhong *et al.* [ZLM07] proposed a multi-core architecture that extended traditional multi-core systems in two ways. Firstly, it provided a dual-mode scalar operand network to enable efficient inter-core communication without using the memory. Secondly, it could organise the cores for execution in either coupled or decoupled mode through the compiler. In coupled mode, the cores executed multiple instructions streams in lock-step to collectively work as a wide-issue VLIW. In decoupled mode, the cores executed independently a set of fine-grain communicating threads that were extracted by the compiler. These two modes created a trade-off between communication latency and flexibility, which could be optimum depending on the required parallelism to exploit. Figure 2.10 presents this multi-core architecture which is named as *Voltron* architecture.

D. Black-Schaffer *et al.* [BSBD+08] analysed a set of IMOs for efficient delivery of VLIW instructions. The evaluation included the cost of memory accesses and wires that were necessary to distribute the instruction bits. The most efficient configuration distributed, indexed, and shared individual memories for the functional units, and used custom circuits to efficiently generate NOPs, resulting in a 56 % decrease in energy and 40 % decrease in area compared to a baseline Filter Cache.

The CLLB (*Clustered Loop Buffer Architecture with Shared Loop-Nest Organisation*) is evaluated in terms of energy budget, required performance, and area cost in Chapter 4. Moreover, a comparison between this architecture and the CELB architecture is performed in Chapter 5 to show which scheme is more suitable for applications with certain behaviour.

2.3.4. Distributed Loop Buffer Architectures with Incompatible Loop-Nest Organisation

With the approach that is described in Section 2.3.3, the gain in efficiency is still limited. In that approach, loops with different threads of control have to be merged (*e.g.*, using loop fusion) into a single loop with a single thread of control. In the case of incompatible loops, the parallelism cannot be efficiently exploited, because they require multiple loop controllers, resulting in loss of performance and area. The use of distributed loop controllers solves this problem. This last architecture is named as DLB (*Distributed Loop Buffer Architecture with Incompatible Loop-Nest Organisation*). References [GMV+04], [JBB+02], and [RLJ+06] are examples of the work done in this class of loop buffer architectures.

M. Jayapala *et al.* [JBB+02] proposed a low-energy clustered IMO for VLIW processors. In this organisation, a simple profile-based algorithm was used to perform an optimal synthesis of the clusters. For a given application, an instruction profile that contained the information of the functional unitsáctivation in every instruction cycle was generated. As there was a direct correlation between the access pattern to the instruction memory and the functional unitsáctivation, this profile could be used to form the clusters. Figure 2.11 shows a template of the proposed architecture in which each L1 cluster had a separate fetch, decode, and issue mechanism. The fetch mechanism across the L1 clusters operated asynchronously, while the issue mechanism was synchronised every instruction cycle. Besides, each L0 cluster had a local controller to regulate the accesses to its corresponding loop buffer. The synchronisation logic of the local controllers caused a performance penalty on the system at the end of every loop iteration. However, simulations indicated that clustering the L0 buffers could reduce up to 45 % of that penalty compared to arbitrarily partitioning.

P. Raghavan *et al.* [RLJ+06] presented a multi-thread distributed organisation that supported execution of multiple incompatible loops in parallel. In the proposed architecture shown in Figure 2.12, each loop buffer had its own local controller, which was responsible for indexing and regulating the accesses. Data sharing and synchronisation were done at the register file level. Due to this fact, context switching and management cost were eliminated, reducing considerably the extra energy cost due to the routing and the interconnect requirements. This enhancement reduced the energy consumed in the IMO and DMH by 70.01 % and improved the performance by 32.89 % compared to SMT (*Simultaneous Multi-Threading*) based architectures.

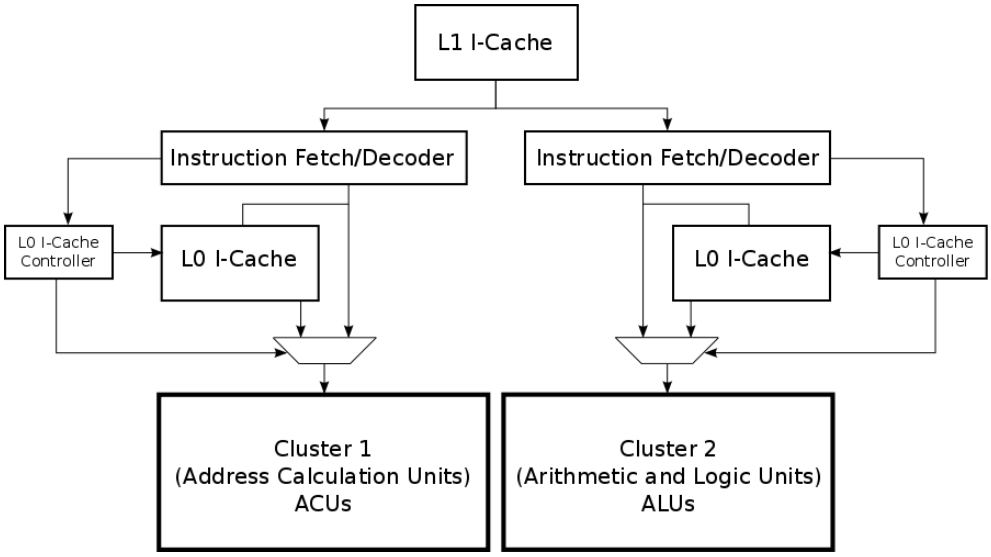


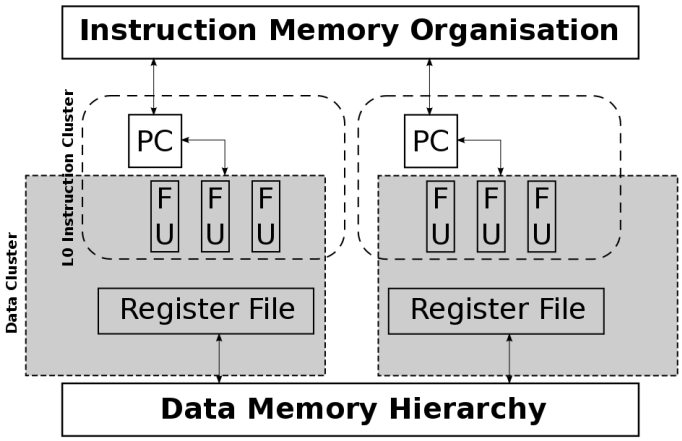
Figure 2.11: Clustered IMO (*Instruction Memory Organisation*) [JBB+02].

J. I. Gomez *et al.* [GMV+04] presented a technique that optimised the memory bandwidth based on the combination of loops with an unconformable header. With this technique, the compiler then better exploited the available bandwidth increasing the system performance.

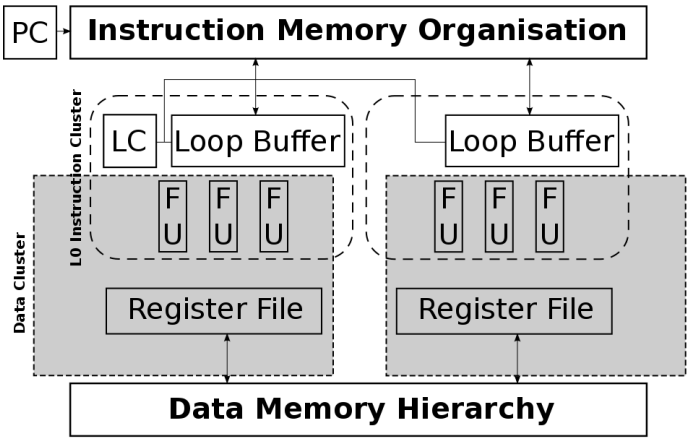
It should be noted that DLB architectures are not always the best in terms of energy efficiency. Depending on the application and using re-configuration of the architecture, an optimal trade-off can be made between multiple and distributed architectures. These facts can be seen in Chapter 6.

2.3.5. Instruction Fetch and Decode Improvements

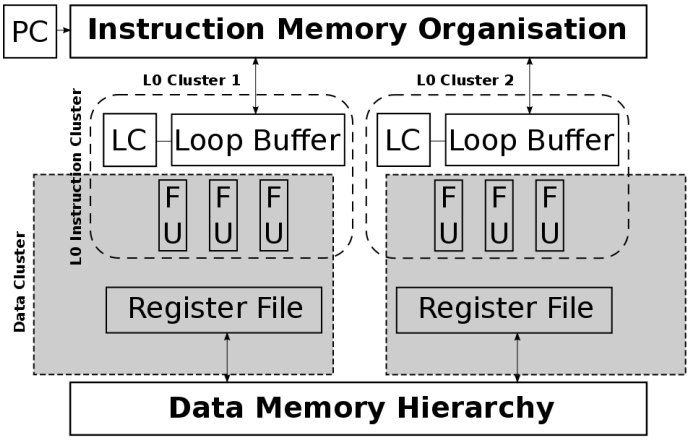
The ISA (*Instruction Set Architecture*) of a DSP typically encodes more operations per cycle than traditional single-issue RISC (*Reduced Instruction Set Computer*) processors. Due to this fact, they require more complex decode and control logic leading not only to more complicated instruction decoders, but also to denser code. Figure 2.13 presents the power consumption breakdown in a DSP core. It is possible to see that in terms of energy distribution over the chip, the control logic and the instruction memories account up to a half of the total consumption. These two modules are among the busiest, and hence, they contribute disproportionately to the total energy consumption of the system. Because instruction fetch and decode logic can consume up to 40 % of the logic, several works have appeared around these specific topics. Next paragraphs describe some references on these fields.



(a) SMT-Based VLIW processor.



(b) VLIW processor with single loop controller.



(c) VLIW processor with distributed loop controller.

Figure 2.12: Architectures supporting multi-threading [RLJ+06].

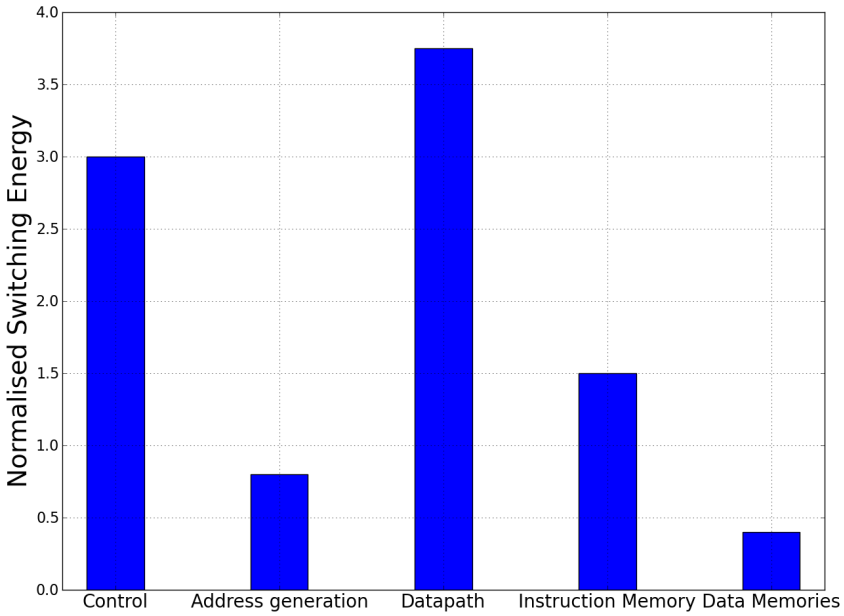


Figure 2.13: Power consumption breakdown of a DSP [BHK+97].

R. S. Bajwa *et al.* [BHK+97] proposed an architectural enhancement that could switch off the fetch and decode logic, if loops could be identified, fetched, and decoded only once. The instructions of the loop were decoded and stored locally, from where they were executed. The energy savings, which could be up to 25 %–30 % of the total energy consumption, came from the reduction of memory accesses as well as the lesser use of the decode logic. The DIB (*Decoded Instruction Buffer*) was a small SRAM memory in which the decoded instructions were written during the first pass through a loop. The decoded instructions were accessed directly from the DIB during subsequent passes through the loop so that, during these subsequent passes, the instruction fetch and decode circuitry could be turned off thereby saving energy. There was an inherent trade-off between the power savings and the size of the DIB. For nested loops, the processor was able to distinguish between loop instructions by using special instructions or by monitoring the program counter. The DIB loading or writing could be initiated by hardware or by the compiler, thus giving programmers the possibility to control the DIB.

W. Tang *et al.* [TGN02] introduced the DFC (*Decoder Filter Cache*). The reduction in the use of the instruction fetch and decode logic was made by providing directly decoded instructions to the processor. A hit in the DFC eliminated one fetch from the cache and the subsequent decode operation, which resulted in energy savings in both instruction fetch and decode stage.

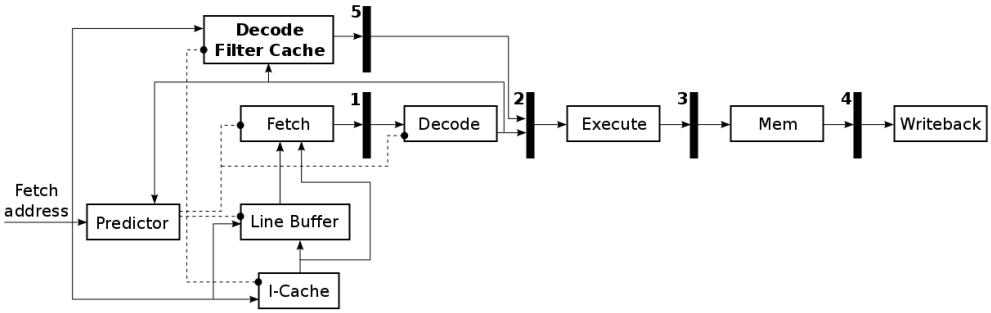


Figure 2.14: Pipeline architecture in a DFC [TGN02].

The key difference between the work done in [KGMS97] and the work proposed in [TGN02] was that, on a miss in the main memory, the missing line was directly filled into the Filter Cache, while in the case of the DFC, the missing line could not be filled into the DFC, due to the fact that the decoded instructions in this line were not available. As a consequence, the DFC could not utilise the spatial locality. To enable energy savings on DFC misses, the author used a line buffer in parallel with the DFC to make use of the spatial locality of the instructions. Figure 2.14 presents the pipeline architecture of this system, showing how this architecture could deliver instructions to the pipeline from the line buffer, instruction cache, and DFC.

2.4. Software Optimisations

The application domain characteristics and the architecture design can be combined. However, the work of the embedded systems designer is not finished yet because the application has to be mapped to the architecture. The performance, the energy consumption, and the behaviour predictability are directly related to the software that is running on the embedded system. Some decades ago, hardware was seen as the only part of the embedded system that controlled the parameters that were mentioned previously. Therefore, every effort to improve the IMO was focused on this part. However, this paradigm shifted and part of the control was moved from hardware to software. The main reasons for this shift were two. Firstly, a design that is based on a simpler and general-purpose processor has less hardware in its critical path. Hence, higher processor speed-ups could be achieved at lower values of energy dissipation. Secondly, hardware components have a view of the application that is based only on its behaviour at run-time. In contrast, compilers have a wider view of the application, because after profiling, it is possible to predict at compile-time the behaviour of the application for a specific input. Due to this advantage, compilers can perform global optimisations in a way that the application can be executed more efficiently. A good example of the benefits

of the efficient control that can be done by software is the management of loop buffers through the compiler. In this scenario, the compiler is responsible for mapping the appropriate parts of the application into the loop buffers activating them only when they are required.

The rest of this Section 2.4 is organised so that each Subsection describes one of the main classes of improvements that are related to the software aspect.

2.4.1. Profiling for Code Optimisation

D. Marculescu [Mar00] addressed the problem of the energy optimisation by using a compiler-assisted technique for code annotation that adaptively selected at run-time the optimal number of instructions to be fetched or executed in parallel. The methodology started simulating the program several times, varying the fetch and the execution rates to obtain the energy consumption of each basic block. With this technique of fine-grain energy characterisation based on profiling, application code could be annotated to enable selection of the optimal number of instructions to be fetched or executed in parallel. Based on experimental results, this approach was up to 23 % better than clock throttling and as efficient as voltage scaling. From this work, it was also possible to see analytically and experimentally that an optimal level of parallelism for energy consumption existed. This trade-off between performance and energy consumption was related to the data-dependency effect, the speculative execution, and the level of parallelism exhibited by common applications.

K. Inoue *et al.* [IMM02] performed a profiling in order to detect unnecessary tag-checks and remove them at run-time in the instruction cache. This architecture exploited execution footprints that were recorded previously in a BTB (*Branch Target Buffer*). The architecture, which was explained previously in Section 2.3.1, had three operation modes: normal mode, omitting mode, and tracing mode. In normal mode, the cache behaved as a conventional instruction cache. During the tracing mode and the normal mode the cache performed tag-checks. Nevertheless, it was only in the tracing mode when the execution footprints were recorded. After recording them, in the omitting mode they were used to detect whether the conditions for dynamic tag-check omission were satisfied. Figure 2.15 shows an execution example of a loop depicting the behaviour of the extended BTB as well. The performance that was lost in order to achieve a reduction in energy consumption was due to the operation in tracing mode. When the architecture was in this mode, the writing of the execution footprints required one processor-stall cycle. Besides, there was an energy consumption overhead related to the action of saving the address in the PBAreg, which was compensated by the energy reductions that were done by removing tag-checks.

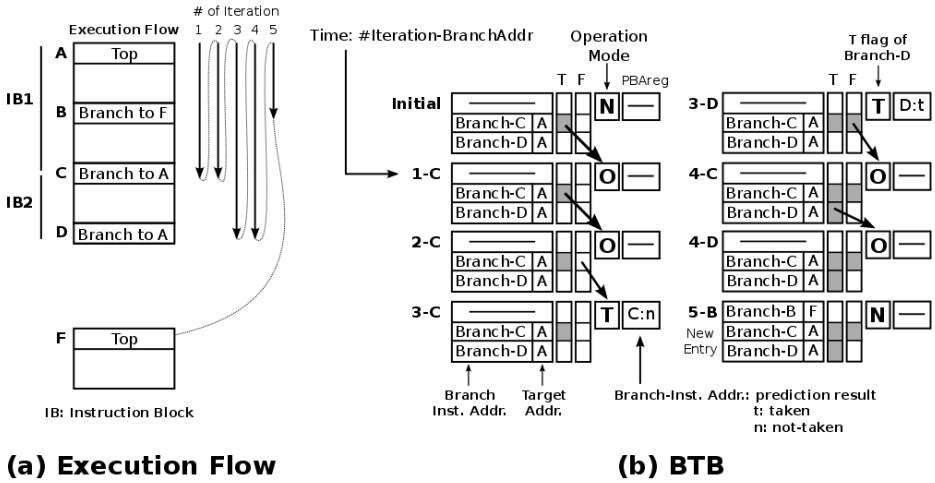


Figure 2.15: Operation example of the HBTC (*History-Based Tag-Comparison*) implementation scheme [IMM02].

2.4.2. Source Code Transformations

Code transformation is the operation that takes an application code and generates another application code, avoiding any modification in its behaviour and serving to optimise it for production cost, area, performance, or energy consumption.

In the work that is presented in reference [IY00], a technique was presented which merged frequently executed sequences of object codes into a set of single instructions. With this idea, the number of accesses to the main memory was dramatically reduced. The merged sequences of object codes were restored by an instruction decompressor before decoding the object codes. However, merging too many sequences into a single instruction led to increase the energy consumption in the decompressor. This technique extracted an optimal selection of code for energy consumption based on two strategies: a packing approach for basic blocks and a sequence merging approach. The packing approach was based on the correlation of basic blocks with a special instruction. The sequence merging was based on the relationship between a sequence of several basic blocks and the special instruction that represented them. This technique had an energy reduction of more than 65 % in the best case compared to an IMO without this enhancement.

N. D. Liveris *et al.* [LZSG02] proposed a flow that iteratively applied source code transformations to improve performance in the instruction cache. The

procedure was driven by a set of analytical equations that predicted the number of misses based on the parameters that were related to the application code and the cache structure. Two high-level code transformations were proposed for this purpose: loop-splitting and function call insertion. Both transformations were aimed at the optimisation of code located in the scope of loop-nest. On the one hand, loop-splitting improved locality of the references of the instruction cache, if the code that was enclosed by a loop-nest was distributed after the application of the transformation. On the other hand, with function call insertion, rarely executed code segments in a loop-nest were replaced by function calls. Besides, an insight analysis, which resulted in the formulation of the analytical equation that was used to predict the number of misses in the instruction cache, was presented.

2.4.3. Mapping

The appropriate mapping techniques for the DMH are applied first in order to detect the bottleneck of the IMO.

K. Pettis *et al.* [PH90] presented algorithms of code positioning. Profiling data was used as an input for the compiler to reduce the overhead of the IMO. The first algorithm, that was built on top of the linker, positioned the code based on procedure invocations. The algorithm made use of dynamic call graph information to guide the positioning of the object code of procedures based on the strategy *closest is best*. Figure 2.16 presents an example that clarifies how this algorithm worked. The algorithm increased the chances of having in the same page several procedures causing the reduction of the page working set size. The second algorithm, that was built on top of an optimiser package, positioned code based on the basic blocks that existed within procedures. Groups of basic blocks that exhibited straight-line sequences were identified as chains, and these chains were then ordered according to branch heuristics. An alternative approach was to count the number of times that the control was transferred from one basic block to another. Figure 2.17 depicts an example of basic block structure in which this algorithm gave more importance to the path A-B-D than the path A-C-D. In order to reduce the procedure size, a procedure splitting algorithm was also present. Procedure splitting separated the basic blocks of a procedure into separate regions to minimise the size of the primary procedure. Using this algorithm, it was possible to improve memory page locality, as well as to create an optimal positioning code at the basic block level. As everything was done at compile time, no penalty on execution time was observed. From the results of this work it was possible to see that more benefits can be reached from the basic block level rather than from the procedure level.

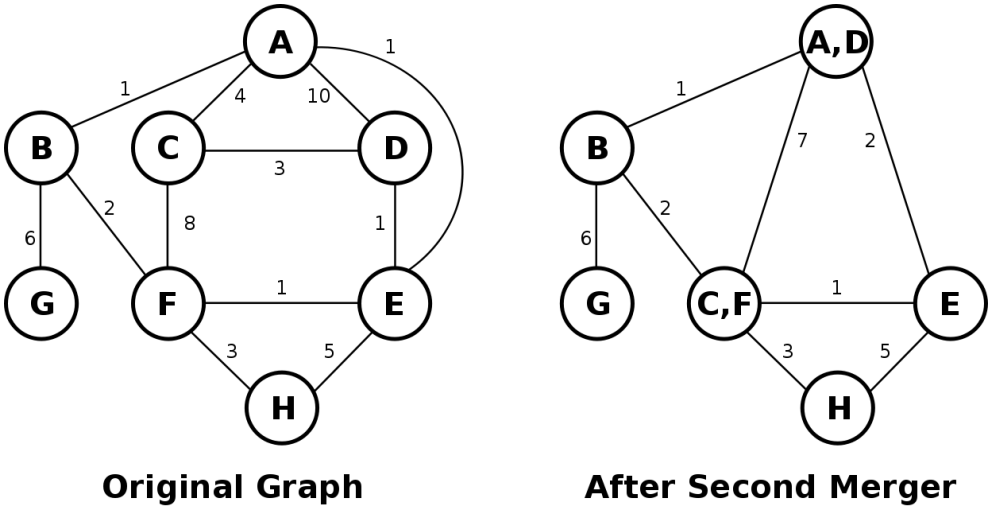


Figure 2.16: Call graph for procedures [PH90].

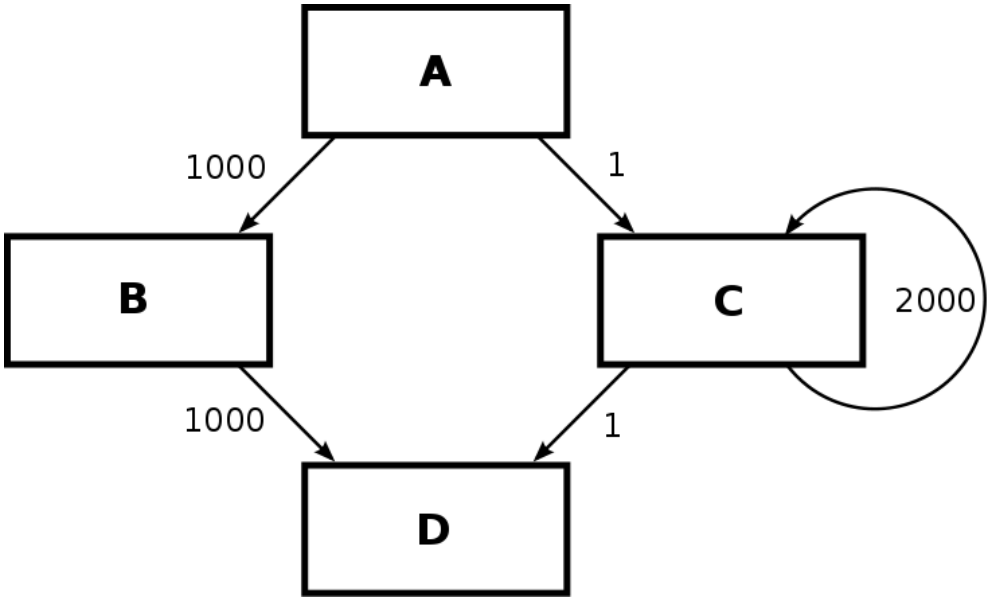


Figure 2.17: Basic block structure with their execution counts in the arcs [PH90].

T. Vander Aa *et al.* [AJB+03] presented a framework to optimise the energy consumption of the loop buffer and the instruction cache in VLIW processors. With loop transformations on multimedia applications, the optimal loop buffer configuration consumed up to 80 % less energy in comparison to a baseline system. If loops were too big to fit into the loop buffer, code transformations were needed not only to increase the loop buffer coverage of the application, but also to optimise the energy consumption:

- **Loop peeling.** Elimination of the conditional test of the loop counter by removing a small number of iterations from the beginning or the end of the loop.
- **Factorisation.** Sometimes the core of multimedia code applications contains multiple similar phases sharing the same code. If this common code is shared in a function, it will be possible to load it only once into the loop buffer. The gain of this optimisation has to be big enough to compensate the penalty that is related to the overhead introduced by the branch to the common code.
- **Loop re-rolling and unrolling loops.** Both reduce the active code size and the energy consumed by the loop buffer that software pipelining introduces when it is used for parallelisation in VLIW processors.
- **Loop splitting.** Reduction of the requirements that are related to the size of the loop buffer by splitting the loop body in two separate loops where one is faster in execution time than the other. The cost of storing and loading the separate loops might be bigger than the gain depending on the granularity of the loop bodies.

2.5. Trends and Open Issues

The existing techniques, that are focused on the reduction of the energy consumption of the IMO and are presented in this related work Chapter, can be classified based on the following main trends:

- The use of small memories in the form of memory banks for the implementation of the L0 instruction cache (*i.e.*, the loop buffer memory).
- The improvement of the efficiency in the partitioning of the L1 instruction cache.
- The addition of enhancements in the compiler to improve the mapping of the application and make an efficient use of the architectural enhancements than can be introduced in the IMO.

During the last decade, the enhancements that were applied in the IMO made use of memory banks to implement the L0 instruction cache in order to reduce the energy consumption of this component of the embedded system. Section 2.3.2 presented the most traditional use of the loop buffer concept: the CELB (*Central Loop Buffer Architecture for Single Processor Organisation*). Because parallelism is a well-known solution to increase performance efficiency, loop transformation techniques were applied to exploit parallelism within loops on single-threaded architectures. However, centralised resources and global communication made these architectures less energy efficient. In order to reduce these bottlenecks, several solutions that used multiple loop buffers were proposed in literature. Section 2.3.3 presented examples of the work done in this field: CLLB (*Clustered Loop Buffer Architecture with Shared Loop-Nest Organisation*). An efficient parallelism was not achieved with the architectures described previously. Using multiple loop buffer architectures with shared loop-nest organisations, loops with different threads of control had to be merged (*e.g.*, using loop fusion) into a single loop with a single thread of control. In the case of incompatible loops, the parallelism could not be efficiently exploited due to the requirement of multiple loop controllers, resulting in loss of performance. Section 2.3.4 described a new set of architectures that solved this problem: DLB (*Distributed Loop Buffer Architecture with Incompatible Loop-Nest Organisation*).

The work that was done in the partitioning of the L1 instruction cache was based on two ideas. The first idea was the use of different memory implementation methods to implement the components that formed the IMO, such as SRAM-based, register-based, or FF-based memories. Once the implementation method that would be used by each component was selected, the code of the application could be split between the different components trying to be as much efficient as possible from the energy consumption point of view. The second idea comprised the study of the number and the size of the partitions that were optimal for a given goal (*e.g.*, performance, access time delay, energy consumption). Most approaches that were related to cache memories assumed that this automated tuning was done statically. The tuning was done only once during design time, and was based on the designer's performance constraints and application characteristics. However, other cache tuning approaches could be used dynamically, while an application was being executed on the microprocessor. It can be concluded that both ideas of partitioning the L1 instruction cache were not exclusive, because they could also be applied together and at the same time.

In the research work, that was related to the enhancements in the compiler for the improvement of the application mapping, the main goal was to reduce the performance and the energy penalties that were related to the

instruction cache misses. Using access patterns in algorithms, the IMO could be partitioned into multi-banks which could be accessed independently. The algorithm was who decided which part of the application code was stored in each one of the components that formed the IMO. Apart from solving these issues, code transformations like code scheduling, loop unrolling, loop morphing, and software pipelining could reduce the number of unnecessary accesses that the IMO could suffer. Besides, compilers had an important role in the energy management of banked memories. At compile-time and after scheduling the instructions on the memory space, the operating modes of the banks could be used efficiently to reduce the energy consumption by being assigned to different segments of the code.

From the literature study that was presented in this Chapter, it is possible to see that future research on IMO will be focused on the development of enhancements and optimisations that increase the exploitation of parallelism in architectures not only to improve the performance, but also to become more energy efficient. In order to achieve this purpose, future designers of IMOs will have to keep in mind that an increase in the parallelism of the system can be directly related to an increase in performance, but not necessarily in energy efficiency. For instance, heavily partitioned IMOs are better in energy efficiency than centralised IMOs as shown by the results that were previously exposed in the analysis of the efficient partitioning of the L1 instruction cache. Nevertheless, due to the overhead that is introduced in address decoding and periphery, heavily partitioned IMOs may have a worse delay and hence longer clock cycle. Therefore, it can be concluded that the more centralised is the IMO, the more reduction in the delay can be achieved. This fact is true, only if the memory is internally optimised. However, these architectures can be worse in the energy consumption per memory access. The design/compilation complexity will be also taken into account in future designs of the IMO. As could be seen, DLB architectures were the best in performance due to the management of incompatible loop-nests. However, the design complexity of this kind of architectures was high. In this case, due to this drawback, researchers will use co-design to make an optimal trade-off between CLLB and DLB architectures. This example provides a nice illustration of the trade-off that exists between performance and design/compilation complexity. The concern on the design of the IMO is appearing slowly due to the fact that electronic systems are starting to be characterised by restrictive resources and low-energy budgets. Therefore, it will be crucial to introduce any enhancement in the IMO to allow not only to decrease the total energy consumption, but also to have a better distribution of the energy budget throughout the embedded system. Despite that the *Core 2* architecture [INT11] has the only architectural enhancement that is starting to be applied, the number of implementations on commercial devices of the enhancements that were described in this Chapter will be increased considerably in the coming future.

2.6. Related Work and Contribution

This Chapter has as target to provide an up-to-date picture of the current status of the research performed in the IMOs from the energy consumption point of view. Previous research has pointed out that the energy-inefficient memory architectures are the bottleneck of the contemporary embedded systems. Besides, the software that is running on them is becoming increasingly complex. M. Verma *et al.* [VM07] carried out an exploration of the IMO, as well as an evaluation of the software optimisation techniques for the optimal utilisation of each one of them. Using real-life benchmarks, M. Verma *et al.* [VM07] showed that linking memory architecture design with architecture aware compilation resulted in fast, energy-efficient, and timing predictable memory accesses.

Examples of surveys that deal with the architectural aspect are [PV97], [Sch02], and [dAEES97]. M. Pedram *et al.* [PV97] presented a wide and detailed coverage of the issues that designers face at the physical level of design abstraction. The main claim of this work was that the design had to be optimised not only for energy, but also for performance and area. J. P. Scheible [Sch02] presented a survey of the storage options and configurations that could be used for the implementation of IMOs. This work outlined the comparative advantages, drawbacks, and trade-offs of the storage options and configurations, helping to choose the right one for the implementation based on the specific and required characteristics. Focused on the ROM (*Read Only Memory*), E. de Angel *et al.* [dAEES97] presented a survey of low-energy techniques at the circuit and the architecture level. This work showed that the efficiency of the techniques depended on the data that was stored in the ROM core, the speed requirements, and the area overhead.

References [PCD+01], [WK03], and [Ace02] provide a good overview of the optimisation techniques that are related to the software aspect. P. R. Panda *et al.* [PCD+01] presented techniques that were used in optimisations of the DMH in embedded systems. Topics as code transformations and memory addressing were discussed in detail based on an original classification: platform-independent memory optimisations that operated on a source-to-source level, and platform-dependent optimisations that could be applied to memory structures at different levels of architectural granularity. However, the drawback of this work was that the context of parallel platforms, such as task-level parallelism, has not been addressed. W. Wolf *et al.* [WK03] surveyed embedded software techniques showing that software design and compilation could take advantage of the fact that the hardware target was known, and the fact that it was possible to spend more time and computational effort in order to improve characteristics such as performance, power, and manufacturing cost. O. Acevedo [Ace02] showed the importance of each kind of instruction

on the energy consumption of the system. As each instruction of a given program activated specific parts of the microprocessor, the election of the correct instruction could generate a reduction of the energy consumption.

However, survey papers that combine both the analysis of software and hardware optimisations provide the best overview. L. Benini *et al.* [BMP03] presented a general overview of the energy-aware design of both the IMO and the DMH through different memory technologies, architectures, and optimisations that were applicable at various levels of abstraction. Only one drawback was found in this work, the study of the IMO was not deep enough, because important issues lacked in the analysis of the implementation and behaviour of the IMO.

Also, it is possible to find surveys that are focused on specific fields. For instance, G. F. Welch [Wel95] addressed the question of which techniques could be employed in computer operating systems to reduce the energy consumption of today's mobile computing devices. W. Chedid *et al.* [CY02] described different management techniques aiming to reduce energy consumption in computer systems. I. Marin *et al.* [MAZA05] showed that energy-aware design for sensor nodes was not a trivial task. These sensors only had as energy supply a pair of batteries that must let them live up to five years without substitution. That is why it was necessary to develop energy-aware algorithms that saved battery lifetime as much as possible.

Apart from providing an up-to-date picture of the current status of IMOs, and giving to the reader a first grasp on the fundamental characteristics and design constraints of various types of IMOs, the work that is presented in this Chapter attempts to provide a complete literature study of the research activities that are related to the IMO, and to serve as a guideline in our objective to provide an IMO with a low-cost energy per task. As it is possible to see from this Section, previous research presented different explorations and evaluations of novel memory hierarchies and software optimisation techniques. Nevertheless, hardware and software optimisations were presented isolatedly in most of the cases. L. Benini *et al.* [BMP03] was the only survey work that presented different memory technologies, architectures, and software optimisations describing how each one of them affected the others. The research lines that are related to the IMO are analysed deeper in this Chapter than in [BMP03], because this Chapter is focused only on the IMO, outlining the future trends, evolutions, and challenges of the IMO. This has allowed the development of a high-level energy estimation tool that explores the architectural implementations, compiler configurations, and code transformations that are related to the IMO (see Chapter 3). Besides, thanks to this energy estimation tool, it has been possible to perform high-level analyses

not only of the promising loop buffer schemes (see Chapter 4), but also when these are applied in real-life embedded applications (see Chapter 5). Based on previous analyses, it has been possible to present a high-level trade-off analysis of the loop buffer implementations to show the correct process design that embedded systems designers have to follow in order to have an efficient loop buffer architecture for a certain application (see Chapter 6).

In the next chapter...

the reader will find a description of a high-level energy estimation tool that, for a given application and compiler, allows the exploration not only of architectural and compiler configurations, but also of code transformations that are related to the instruction memory organisation for embedded systems.

Chapter 3

IMOSIM: Exploration Tool for Instruction Memory Organisations based on Accurate Cycle-Level Energy Modelling

"Technology is nothing. What's important is that you have a faith in people, that they're basically good and smart, and if you give them tools, they'll do wonderful things with them."

— Steven Paul Jobs (Steve Jobs).

Due to the fact that the design space of the enhancements for reducing the energy consumption of the instruction memory organisation is huge, this Chapter proposes a high-level energy estimation tool that, for a given application and compiler, allows the exploration not only of architectural and compiler configurations, but also of code transformations that are related to the instruction memory organisation. The proposed tool, with a mean error of 3.95 %, achieves reductions in time and effort to explore the design space of the instruction memory organisation.

3.1. Introduction and Related Work

Embedded systems demand different hardware architectures to run applications that range from multimedia consumer devices to industry control systems. However, unlike general-purpose computer systems, embedded systems have to provide high-computation capability, reliability, predictability,

and meet real-time constraints with not only limited resources, but also a low-energy budget. The combination of previous requirements and constraints makes the reduction of total energy consumption become a big challenge for embedded systems designers. For more information see Section 1.1.1.

As shown in Section 1.2, a typical embedded system is composed of a processor architecture, a DMH (*Data Memory Hierarchy*), an IMO (*Instruction Memory Organisation*), and an inter-core communication network. For more details see Figure 1.4. Research works like [CRL+10] and [VM07] demonstrated that both memory devices not only take significant portions of chip area, but also now account for up to 40 %–60 % of the total energy budget of an embedded instruction-set processor platform (see Figure 1.7). The energy consumption of the IMO is not negligible and needs to be optimised in order to reduce the overall energy consumption of the embedded system. In order to reduce the energy consumption of the IMO, embedded systems designers modify and/or partition the IMO. On the one hand, loop buffering is a good example of effective scheme for the modification of the hierarchy that exists in the IMO (see Figure 3.1). J. Kin *et al.* [KGMS97] showed that storing small program segments in smaller memory (*e.g.*, in the form of a LB (*Loop Buffer*)), the dynamic energy consumption of the system was reduced significantly. On the other hand, banking is a good example of effective method for the partitioning of the IMO [KKK02] (see Figure 3.2). Apart from the possibility of using multiple low-power operating modes, the use of banks reduces the effective capacitance as compared to a single monolithic memory, which leads to further energy reductions [BMP00, FEL01, LK04]. Section 2.3 discusses extensively these hardware optimisations of the IMO.

The ITRS *(International Technology Roadmap for Semiconductors)* [ITR12] predicts that the energy consumption of embedded systems will continue its fast growth for the next decade due to their ever-increasing complexity and size. Apart from the energy consumption, reliability and predictability are also becoming critical, because both of them are directly related to the energy consumption and its distribution over the system. Due to these issues, a high-level energy estimation tool is required during the design process of an embedded system to increase the simulation speed and the energy savings. Accurate energy models have to be created for each instance that composes the IMO, in order to estimate at high level the total energy consumption of the IMO. The energy estimation is performed based on simulations that take into account the integration of the energy models of every instance that composes the IMO. Previous works have showed sophisticated energy modelling methods to precisely estimate the energy consumption at system level [BLRC05], [ANMD07], [KAA+07], and [LKY+06]. However, these methods lack a design space exploration of the different IMO architecture

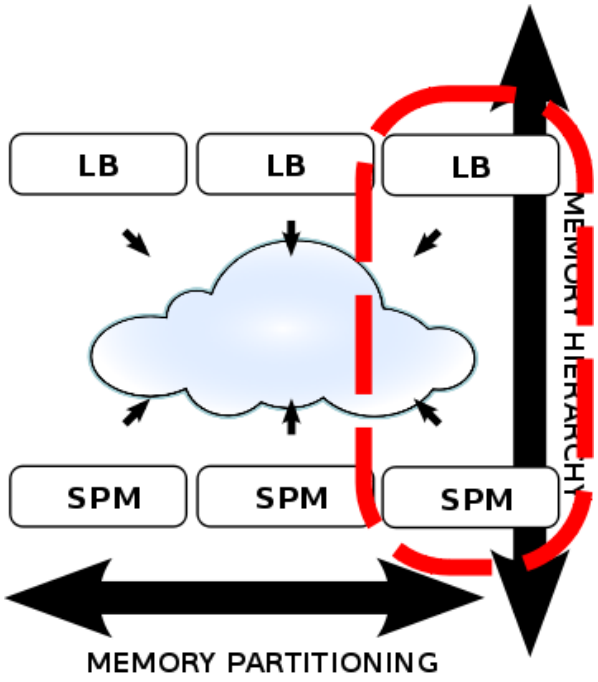


Figure 3.1: Memory hierarchy in the instruction memory organisation.

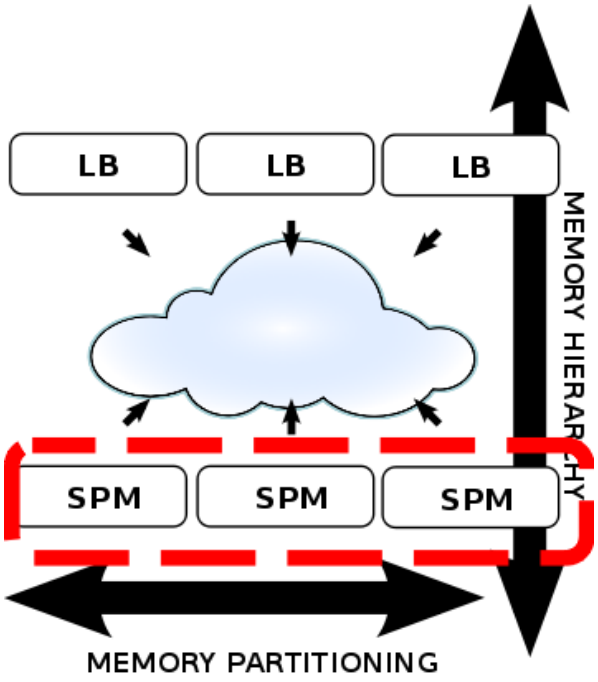


Figure 3.2: Memory partitioning in the instruction memory organisation.

options from the energy consumption point of view. In some cases, previous works use relatively basic IMOs (*e.g.*, based on L1 instruction caches) avoiding the more complex IMO architecture options that are potentially much more energy-efficient. In other cases, the presence of the IMO in the system is completely ignored, because its energy consumption is counted as a part of the energy consumption that is assigned to the operations that are executed on the processor architecture. This Chapter proposes a high-level energy estimation and exploration tool that explores the different architectural and compiler configurations that can implement the IMO. Figure 3.3 shows the block diagram of the proposed high-level energy estimation and exploration tool. As can be seen in this Figure, this tool automatically processes a given application based on its characteristics, a given processor architecture based on its requirements, and a given power consumption library of different memory instances, in order to help embedded systems designers to find the optimised configuration of the IMO for the total energy consumption of the embedded system. The proposed tool, with a mean error of 3.95 %, achieves significant reductions in time and effort to explore the design space of the IMO.

3.2. Design Space of the Instruction Memory Organisation

The work that is presented in this Chapter is based on an architectural classification that contains three major scenarios where energy-efficient loop-oriented applications and platforms can fit in. In the following paragraphs, the representative architecture of the IMO of each scenario is explained in detail.

The CELB (*Central Loop Buffer Architecture for Single Processor Organisation*) represents the most traditional use of the loop buffer concept. For more details see Section 2.3.2. Figure 3.4 shows the generic architecture of this IMO, which neither has partitioning in the loop buffer architecture nor in the PM (*Program Memory*), and its connections depend on a single centralised component. Therefore, parallelism in the execution of an application cannot be achieved in this architecture.

The centralised resources and global communication of single-threaded architectures make CELB architectures less energy efficient, when techniques of loop transformations are applied to exploit parallelism within loops. The CLLB (*Clustered Loop Buffer Architecture with Shared Loop-Nest Organisation*) mitigates these bottlenecks. For more details see Section 2.3.3. Figure 3.5 shows the CLLB architecture, which inner connections are controlled by one single component. In this case, the controller is more complex, because it controls the partitions that exist in the loop buffer architecture and program memory.

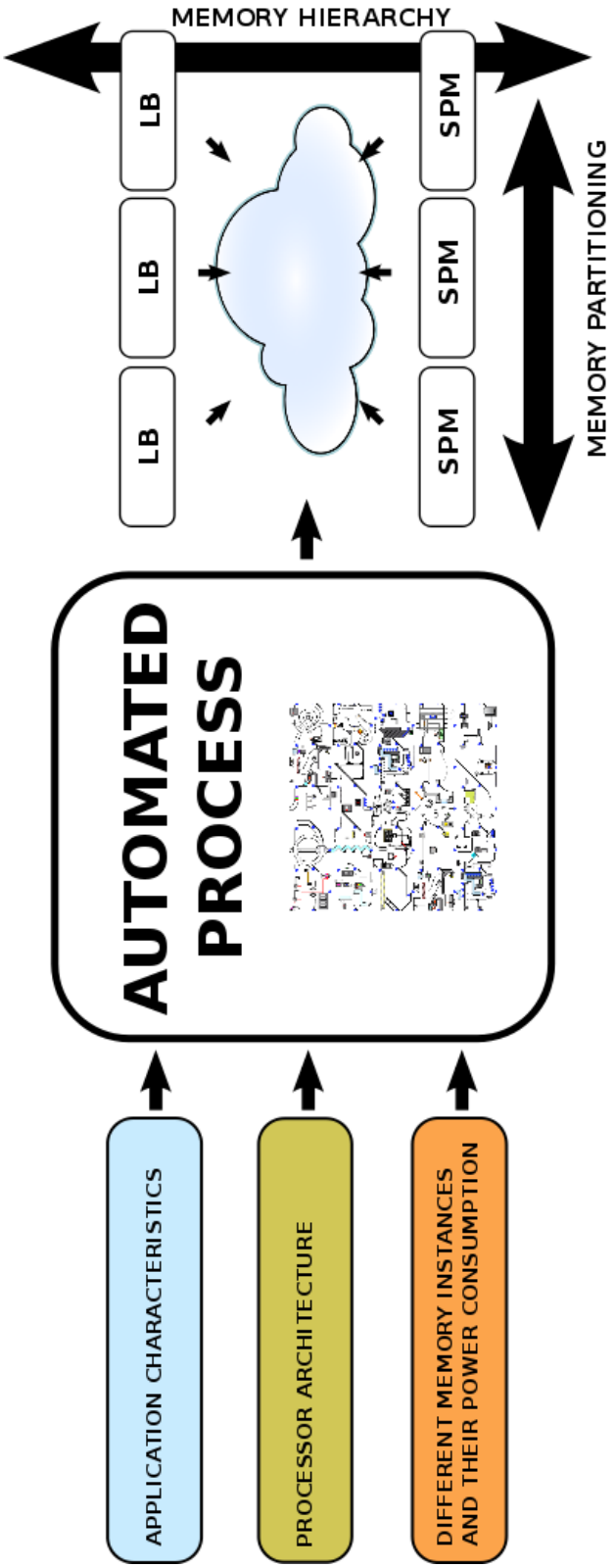


Figure 3.3: Block diagram of the high-level energy estimation and exploration tool.

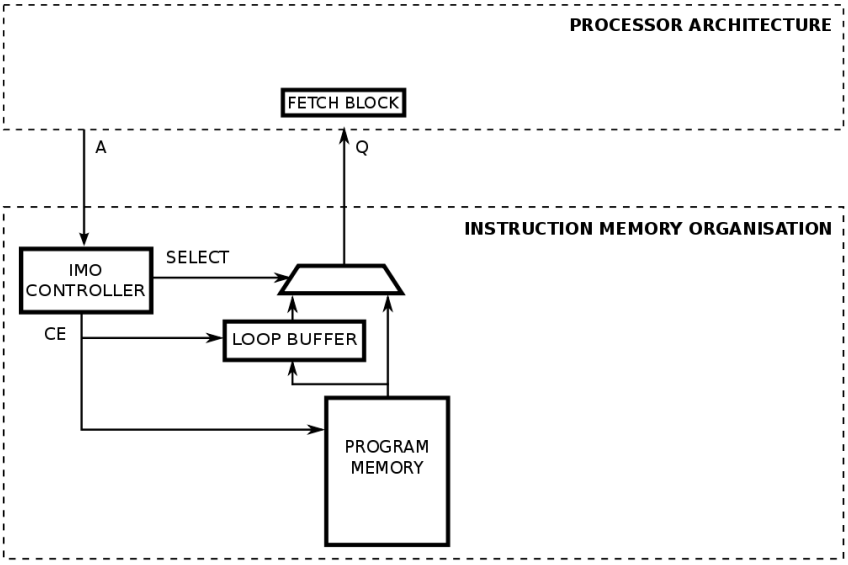


Figure 3.4: Instruction memory organisation with a central loop buffer architecture for single processor organisation.

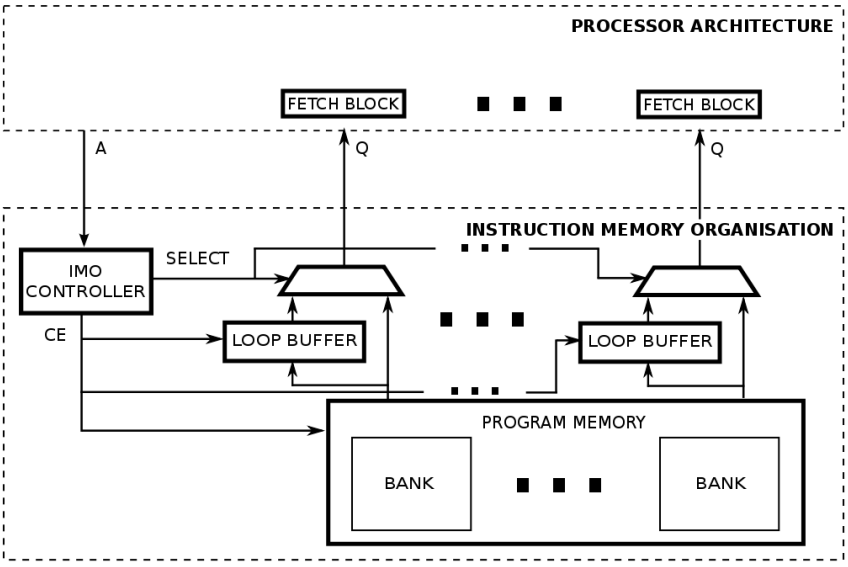


Figure 3.5: Instruction memory organisation with a clustered loop buffer architecture with shared loop-nest organisation.

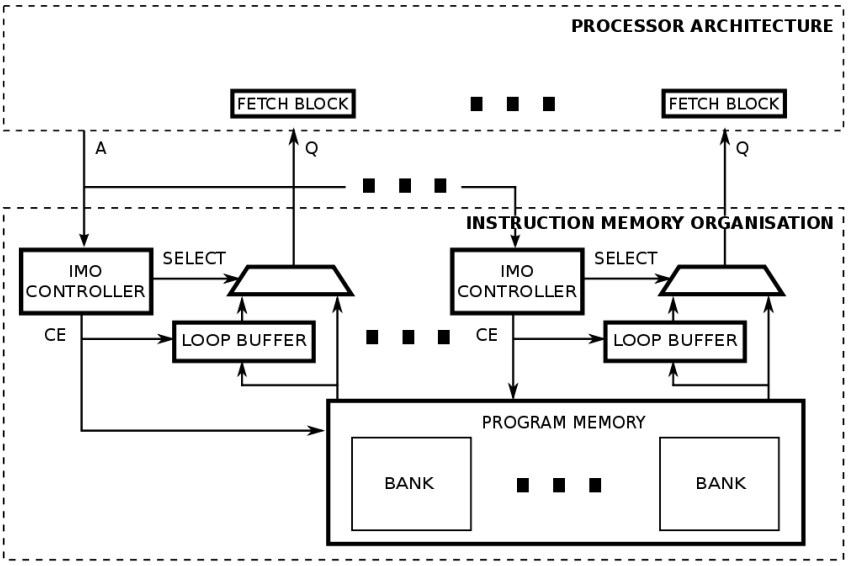


Figure 3.6: Instruction memory organisation with a distributed loop buffer architecture with incompatible loop-nest organisation.

Efficient parallelism is not achieved with a CLLB architecture, due to the fact that loops with different threads of control have to be merged into a single loop with a single thread of control. The required code transformation is performed using techniques like loop transformations (*e.g.*, loop fusion). However, not all loops of an application can be efficiently exploited in this manner. In the case of incompatible loops, parallelism cannot be efficiently exploited, because they require multiple loop controllers, which results in loss of energy and performance. Therefore a need exists for a multi-threaded platform, that could support execution of multiple incompatible loops with minimal hardware overhead. This problem is solved by a DLB (*Distributed Loop Buffer Architecture with Incompatible Loop-Nest Organisation*). In this loop buffer architecture, not only the loop buffer memories are distributed across the IMO, but also the loop buffer controllers that manage them. Therefore, in this special set of loop buffer architectures, each loop buffer memory has its own local loop buffer controller. Due to this fact, DLB architectures can work like multi-threaded platforms allowing the execution of incompatible loops in parallel with minimal hardware overhead. For more details see Section 2.3.4. Figure 3.6 shows the generic architecture of this recent representative loop buffer architecture. As shown in this Figure, the inner connections of this IMO are managed by a logic of controllers that is distributed across the architecture. In this IMO, partition exists in both the loop buffer architecture and the program memory. The controller of this

IMO is even more complex than the controllers of the previous ones, because it also controls the execution of each loop in the corresponding loop buffer architecture to allow the parallel execution of loops with different iterators. The energy estimation and exploration tool that is proposed in this Chapter is the only one, known by the author, that allows the simulation of this novel IMO from the energy consumption point of view.

3.3. IMOSIM (*Instruction Memory Organisation SIMulator*)

IMOSIM (*Instruction Memory Organisation SIMulator*) is a high-level energy estimation tool that, for a given application and compiler, explores different architectures and configurations that can compose the IMO. This exploration helps embedded systems designers to find the architectural and compiler configuration that is optimal from the point of view of the total energy consumption of the embedded system. In order to perform a complete design space exploration of the IMO, the representative architectures that are shown in Section 3.2 are used to mimic every loop buffer architecture that is already published in literature. As shown in Figure 3.7, IMOSIM requires three inputs in text file format to know the requirements of the embedded system architecture and the embedded application under simulation: the store access history report of the application, the requirements of the embedded system, and the cycle-level energy models of each one of the components that compose the representative IMOs that are described in Section 3.2. The user has as output not only the text files that contain the energy profiles of the components that form the representative IMOs, but also the graphs that represent these data. Figure 3.8 is a good example of graph that embedded systems designers can get from IMOSIM.

The description of the system is supplied to IMOSIM through the information of the system requirements and the store access history report. The system requirements information is a complete description of the interface between the processor architecture and the IMO. This input provides the width and number of instructions that the processor architecture requires in order to execute the given application, as well as the frequency that is used in the system. Therefore, this input provides to IMOSIM the description of the inter-core communication network that exists in the system. The store access history report provides to IMOSIM the profile information that is related to the run-time behaviour of the application. This input information, that is generated by the same ISS (*Instruction Set-Simulator*) that is used to corroborate the correct functionality of the system, contains the memory addresses that are accessed by the processor architecture in each execution cycle.

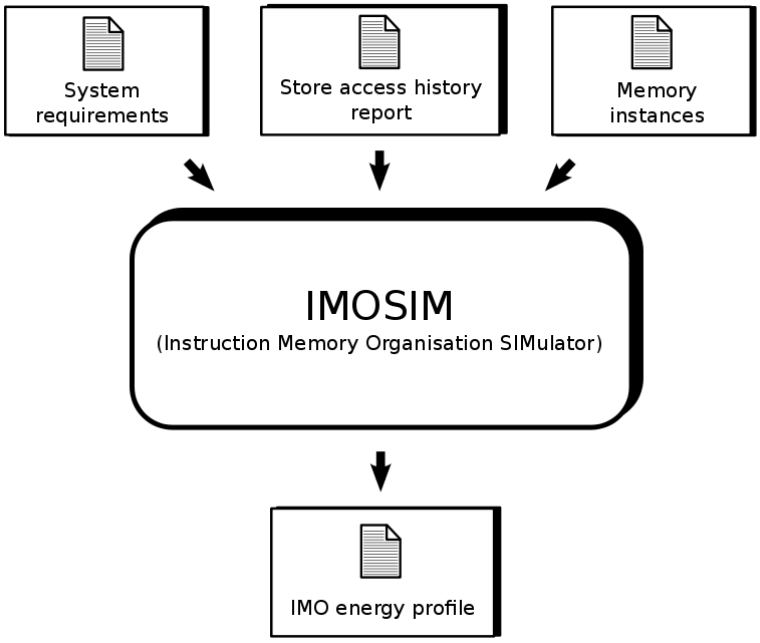


Figure 3.7: Flowchart of IMOSIM (*Instruction Memory Organisation SIMulator*).

The description of the memory instances that can compose the IMO is supplied to IMOSIM through cycle-level energy models. These models can be obtained in two ways. Either from the data sheets of the commercial memories that the embedded systems designer wants to use, or creating the energy models of the memory instances from post-layout back annotated energy simulations. In this last option, RTL (*Register-Transfer Level*) simulations have to be performed to produce accurate energy waveforms of the memory instance under modelling with some user-specified training VCD (*Value Change Dump*) files. Therefore, the energy consumptions, that are related to the different kind of accesses that the memory instances have in their diverse operation modes, are known through trend-line equations that are obtained from discrete measurements by polynomial curve fitting. The degree of the polynomial equation is chosen by the embedded systems designer based on the desired level of accuracy. In order to cover the whole design space of a memory instance, variations in depth, width, and technology used in its implementation are done. Although, the energy models of the memory controllers contain the control logic of the connections between different instances and the activation of the different operation modes, these models are directly related to the parameters that define the memory array. Hence, the same procedure is followed to derive the equations that describe the energy consumption of this

part of the IMO. IMOSIM models the memory controllers of the loop buffer architectures as follows:

- **CELB architectures.** The logic of the memory controller controls the connections to activate the whole loop buffer architecture when the execution of a loop is detected, as well as the banks that can compose the program memory.
- **CLLB architectures.** The memory controller of these representative architectures increases its logic to control also the activation of the loop buffer memories that are inside of the loop buffer architecture.
- **DLB architectures.** Apart from the previous capabilities, the memory controller of this architecture is able to control every loop buffer memory of the loop buffer architecture isolately. This fact allows the parallel execution in this loop buffer architecture of loops with different iterators and body size.

IMOSIM processes its three inputs based on mathematical equations that take into account the state of the IMO in the specific execution cycle that is evaluated. The loop buffer configuration is assumed to be fixed during the simulation. Due to this constraint, a limited set of splits can be chosen for a specific loop. A split of the input loop is defined as a vector $S_t \; \epsilon \; \Re^n$, where $(S_t)_i$ is the number of instruction words that are stored in bank i at time t, and n is the number of banks that form the loop buffer memory. Equation 3.1 describes formally the range of sizes that a split can have. The size of each split is assumed to be continuous and ranges from zero to the value of the size of the biggest bank (B_{max}).

$$0 \leq S_t \leq B_{max} \quad \forall t \tag{3.1}$$

In the energy models, the size of each bank that forms the loop buffer memory is defined as a vector $B_i \; \epsilon \; \Re$, where B_i is the size of the bank i. The sizes of the banks are assumed to be continuous and range from zero to a maximum size value B_{max} as it is described in Equation 3.2. The value B_{max} as well as the size of each bank are selected to keep the addressing logic as simple as possible.

$$0 \leq B_i \leq B_{max} \quad \forall i \tag{3.2}$$

The accesses are defined as a vector $A_t \epsilon \Re^n$, where $(A_t)_i$ is the number of accesses to the bank i at time t, and n is the number of banks that form the loop buffer memory. The relation between the accesses to a memory bank $(A_t)_i$ and the possible splits of the input loop $(S_t)_i$ is expressed by Equation 3.3. From this Equation, it is possible to derive that $A \subseteq S$.

$$A_t = S_t \iff (S_t)_i \leq B_i \quad \forall i \tag{3.3}$$

The total energy consumption of each one of the memory instances that compose the IMO is defined, as shown in Equation 3.4, as a vector $E_i \epsilon \Re$, which is modelled as summation of dynamic energy consumption and static energy consumption. On the one hand, the dynamic energy consumption $(E_{dynamic})_i$ depends on the kind of access $(A_t)_i$ that is performed in the memory instance B_i. On the other hand, the leakage energy consumption $(E_{leakage})_i$ is composed of the energy that is consumed in the operating modes that are activated during the execution of the cycle. Dynamic and leakage energy consumption are described by Equation 3.5 and Equation 3.6, where $(A_t)_i$ is a vector that indicates which kind of access is being performed in the memory instance, and C_t is a vector that indicates the operation mode (*active*, *off*, and *retention*) of the memory instance for the specific execution cycle in which the access to the memory instance is being performed. The output of IMOSIM is the evaluation of Equation 3.4, Equation 3.5, and Equation 3.6 for each execution cycle of the application under simulation.

$$E_i = (E_{dynamic})_i + (E_{leakage})_i \tag{3.4}$$

$$(E_{dynamic})_i = (E_{write})_i(A_t)_i + (E_{read})_i(A_t)_i \tag{3.5}$$

$$(E_{leakage})_i = ((E_{act})_i + (E_{off})_i + (E_{ret})_i)C_t \tag{3.6}$$

For ultra-low energy embedded systems, the temperature variations that are caused by the running of these systems are negligible. Besides, the maximum temperature, that these systems can achieve, is lower than the temperature in which reliability failures can appear. Due to the fact that the current realisation of IMOSIM is focused on ultra-low energy embedded systems, leakage-based thermal dependency and PVT (*Process, Voltage, and Temperature*) variations do not need to be considered yet by this version of IMOSIM.

3.4. Experimental Results

In this experimental framework, the processor architecture is designed using *Target Compiler Technologies* [TAR12]. For more information see Appendix B. Therefore, in order to corroborate the correct functionality of the system as well as generate the store access history report, the ISS of *Target Compiler Technologies* is used. The evaluation is performed using TSMC (*Taiwan Semiconductor Manufacturing Company*) 90nm LP (*Low Power*) libraries and commercial memories. A clock frequency of 100MHz is selected. Real-life embedded applications in the ultra-low energy domain are used as benchmarks to evaluate the high-level energy estimation and exploration tool that is proposed in this Chapter. The selected benchmarks, which can be seen in Table 3.1, are prime examples not only of all application domains that are loop dominated, exhibit sufficient opportunity for data and/or instruction level parallelism, comprise signals with multiple word-lengths, and require a relatively limited number of variable multiplications, but also of more general-purpose applications domains that can be found in the area of wireless base-band signal processing, multimedia signal processing, or different types of sensor signal processing. The benchmarks are selected with the goal of showing the great flexibility of IMOSIM.

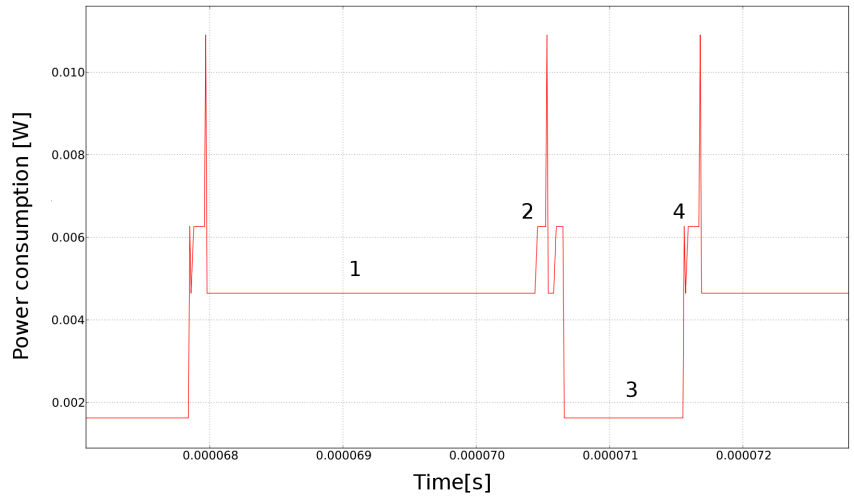


Figure 3.8: Run-time behaviour of IMOSIM showing the power consumption of an IMO based on a CELB architecture.

The run-time output of IMOSIM with the benchmark *AES* (see Section C.3) is presented in Figure 3.8. This Figure shows an overview of how the power

Table 3.1: Profiling information of the benchmarks that are used in the experimental evaluation of IMOSIM.

Benchmark [Reference]	Cycles	Issue Slots	Bits per Instruction	LB Size [Instructions]	Loop Code [%]	NOP Instructions [%]
AES [TSH+10]						
(See Section C.3)	3,347	1	16	32	77.44	0.09
BIO-IMAGING [PFH+12]						
(See Section C.4)	334,071	4	80	64	98.01	26.70
CWT NON-OPTIMISED [YKH+09]						
(See Section C.5)	274,464	1	16	32	6.61	0.48
CWT OPTIMISED [YKH+09]						
(See Section C.6)	102,827	1	20	64	6.36	2.50
DWT [DS98]						
(See Section C.7)	758,216	2	16	518	65.70	25.19
MRFA [QJ04]						
(See Section C.9)	177,170	2	32	64	19.13	17.01

consumption of the IMO changes every cycle depending on the components that are active. In region 1 of this Figure, the instructions are only fetched from the program memory, because during the execution of non-loop parts of the application code, instructions are fetched directly from this memory. In region 2, the IMO detects that a loop is been executed, and therefore, the instructions are fetched from the program memory to both the loop buffer architecture and the processor architecture. In this loop iteration, the loop buffer architecture records the instructions of the loop. Once the loop is stored, the execution of the loop is in region 3, where the instructions are fetched from the loop buffer architecture instead of the program memory. In the last iteration of the loop, which can be seen in region 4, the connection between the processor architecture and the program memory is restored, such that subsequent instructions are only fetched from the program memory.

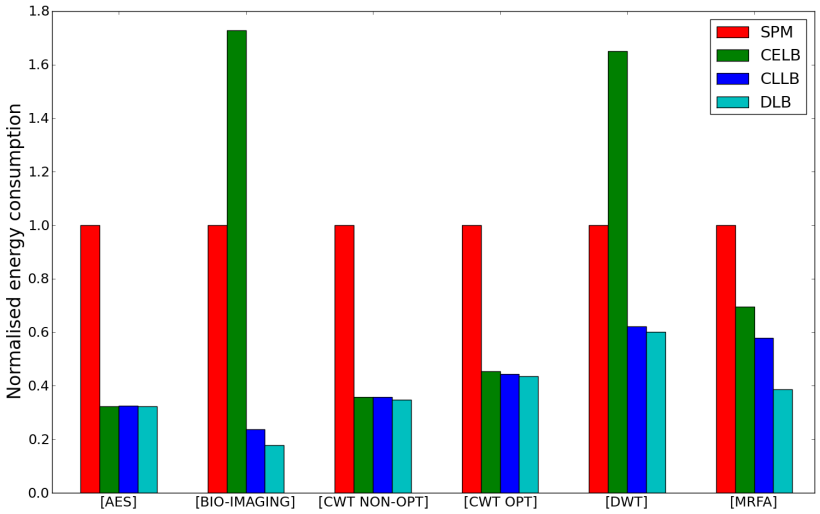
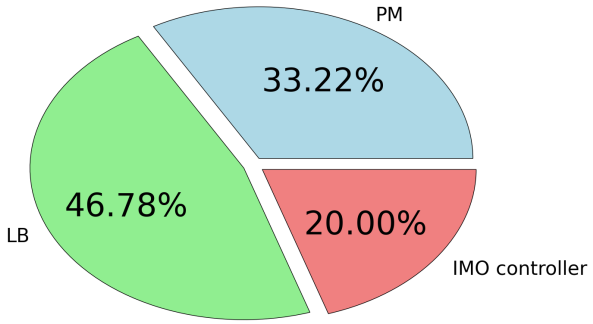


Figure 3.9: Normalised energy consumption in different IMOs running the selected benchmarks.

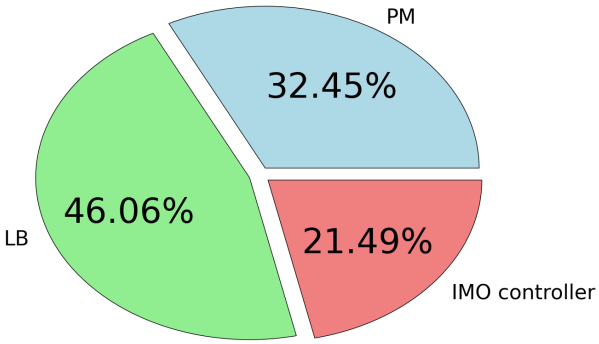
Figure 3.9 contains a bar plot where a comparison is performed among the benchmarks that are presented in Table 3.1. In this Figure, it is possible to see how the code transformations and the options of the compiler affect the energy consumption of the IMO. The baseline architecture is an IMO that is based only on a SPM (*Scratchpad Memory*). As shown in Figure 3.9, not any architectural enhancement that is introduced in the IMO produces a significant reduction of the energy consumption of the system. The reason of the variations in the absolute value of the energy reductions is based on

the percentage of the execution time that is related to loop code. If this percentage is low, the energy savings are smaller than the case where this percentage is high. On the one hand, the difference in energy consumptions between the CELB architecture and the CLLB architecture are related to the loop transformations that are applied in the application in order to parallelise its execution. An effective parallelism leads to higher difference between these two loop buffer architectures, where the CLLB architecture has less energy consumption. However, a poor parallelism will lead to small difference between them, and even, in some cases the CLLB architecture could have more energy consumption than the CELB architecture. On the other hand, the difference in energy consumptions between the CLLB architecture and the DLB architecture is related to the behaviour of the compiler. If the compiler maps the application on the loop buffer architecture effectively, the difference between energy reductions will increase, where the DLB architecture will be the loop buffer architecture that has less energy consumption. These results show that an energy efficient embedded design has to take care not only about the architectural configurations that are used in the embedded system, but also about the characteristics of the application that is running on it.

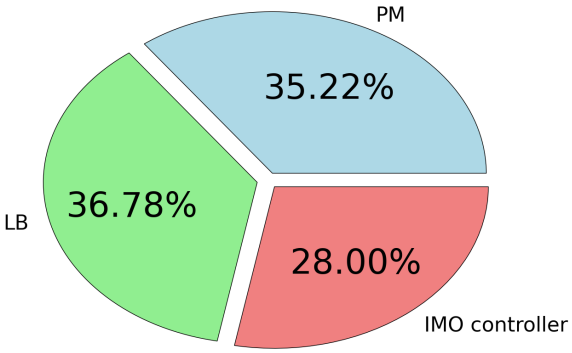
Figure 3.10 shows how for the benchmark *AES* (see Section C.3) the distribution of the energy consumption in the IMO changes from one representative loop buffer architecture to other. If the CELB and the CLLB architectures are compared, it is possible to see that the energy related to the loop buffer controller is increased in the last loop buffer architecture, due to the fact that the controller have to control the activation or deactivation of more components. Also, it is possible to appreciate that the consumption of the loop buffer architecture is reduced. This fact is related to the reduction in the dynamic energy consumption that is caused by the use of smaller memory instances. If the DLB architecture is analysed, it is possible to appreciate that the percentage of the energy consumption that is related to the loop buffer controller of this loop buffer architecture is higher than the loop buffer controllers of the previous loop buffer architectures. This is due to the fact that the loop buffer controller of the DLB architecture is more complex than the previous loop buffer architectures. However, as it is possible to see from this Figure, the efficient management that is performed by the loop buffer controller leads to higher reductions in the loop buffer architecture and, as a consequence in the IMO. It should be noted that the absolute value of the energy consumption of the program memory is not constant in all these IMOs due to two facts. Firstly, the number of cycles that the application requires for its execution over these loop buffer architectures changes. Secondly, the size and width of the memory changes, as well as its composition when it is banked.



(a) CELB architecture.



(b) CLLB architecture.



(c) DLB architecture.

Figure 3.10: Energy consumption breakdown for each one of the representative IMOs.

In order to evaluate the accuracy of this high-level energy estimation and exploration tool, comparisons between IMOSIM and post-layout simulations were performed. The execution on the CELB architecture of the real-life embedded applications that are presented in Table 3.1 shows a mean error of 3.95 %, as it can be seen from Table 3.2. This mean error is compensated by the reductions in simulation time that are offered by IMOSIM, which are based in the advantage of not performing a synthesis of the system architecture every time that this is modified.

Table 3.2: Accuracy evaluation of IMOSIM.

Benchmark [Reference]	**IMOSIM simulation**	**Post-Layout simulation**	**Error [%]**
AES [TSH+10] (See Section C.3)	5.04×10^{-05} [mJ]	4.97×10^{-05} [mJ]	1.34 %
BIO-IMAGING [PFH+12] (See Section C.4)	2.71×10^{-03} [mJ]	2.89×10^{-03} [mJ]	6.79 %
CWT NON-OPTIMISED [YKH+09] (See Section C.5)	4.44×10^{-03} [mJ]	4.35×10^{-03} [mJ]	2.02 %
CWT OPTIMISED [YKH+09] (See Section C.6)	2.08×10^{-03} [mJ]	2.15×10^{-03} [mJ]	3.56 %
DWT [DS98] (See Section C.7)	2.07×10^{-02} [mJ]	2.15×10^{-02} [mJ]	3.76 %
MRFA [QJ04] (See Section C.9)	3.18×10^{-03} [mJ]	3.38×10^{-03} [mJ]	6.23 %

3.5. Conclusion

Researchers have demonstrated that a relevant portion of the total energy budget of an embedded instruction-set processor platform is related to the instruction memory organisation. Previous works showed sophisticated energy modelling methods to precisely estimate the energy consumption of an embedded system. However, these methods lack an energy exploration of the full architecture range of the instruction memory organisation. This Chapter proposes a high-level energy estimation and exploration tool, which finds the optimised configuration for the total energy consumption of the embedded system, by exploring different configurations of the instruction memory organisation, for both given application and compiler. Apart from the reduction in time and effort to explore the design space, the proposed tool allows the exploration of the effects that are caused by code transformations and compiler configurations in the total energy consumption of the instruction memory organisation.

In the next chapter...

the reader will find a high-level analysis of the promising loop buffer schemes that exist for instruction memory organisations in embedded systems. The crucial analysis, that is presented in the next Chapter, proposes a method to evaluate different loop buffer schemes for a certain application, guiding the embedded systems designer to make the correct decision in the trade-offs that exist between energy budget, required performance, and area cost of the embedded system.

Chapter 4

Design Space Exploration of Loop Buffer Schemes in Embedded Systems

"Statistics are like a bikini. What they reveal is suggestive, but what they conceal is vital."

— Aaron Levenstein.

The increasing use of battery powered systems has made the reduction of the energy consumption become an important design goal in the domain of embedded systems. Previous research has pointed out the instruction memory organisation as one of the major sources of energy consumption in embedded systems. In order to decrease this energy bottleneck, the introduction of any enhancement in this component of the embedded system becomes crucial. The purpose of this Chapter is to present a high-level analysis of promising loop buffer schemes that exist in embedded systems. This crucial analysis proposes a method to evaluate different loop buffer schemes for a certain application, guiding embedded systems designers to make the correct decision in the trade-offs that exist between energy budget, required performance, and area cost of the embedded system. Results of this analysis show that, the search of energy savings (up to 76 %) has to take into account the performance and the area penalties, and the impact of the loop buffer implementation technology, in order to choose the most suitable enhancement that has to be introduced in the instruction memory organisation.

4.1. Introduction

Embedded systems not only have a market size that is about 100 times the desktop systems market, but also offer the widest variety of processing power

and cost portion of the electronic market. However, despite this variety, every embedded design can be constrained by the following issues: the required performance, the optimisation of memory area, and the reduction of energy consumption. Among them, the optimisation of the energy consumption is becoming very crucial nowadays not only due to the increasing demand of battery powered systems, but also due to the need of using less expensive packaging. Previous works like [CRL+10] and [VM07] have demonstrated that the IMO (*Instruction Memory Organisation*) takes significant portions of chip area and energy consumption of the embedded instruction-set processor platform.

As shown in Section 1.3, embedded systems designers modify or/and partition the IMO in order to reduce its energy consumption. On the one hand, loop buffering is a good example of effective scheme for the modification of the hierarchy that exists in the IMO. J. Kin *et al.* [KGMS97] showed that storing small program segments in smaller memory (*e.g.*, in the form of a LB (*Loop Buffer*)), the dynamic energy consumption of the embedded system was reduced significantly. On the other hand, banking is a good example of effective method for the partition of the IMO [KKK02]. Apart from the possibility of using multiple low-power operating modes, the use of memory banks reduces the effective capacitance as compared to a single monolithic memory, which leads to further energy reductions [BMP00, FEL01, LK04]. Section 2.3 discusses extensively these hardware optimisations of the IMO.

Due to the fact that the design space of the enhancements for reducing the energy consumption of the IMO is huge, a high-level trade-off analysis of the IMO is crucial and required. The contribution of this Chapter is to present such an analysis between several existing enhancements that are based on the loop buffer concept. This analysis proposes a method to evaluate different promising loop buffer schemes for a certain application, helping embedded systems designers to take decisions in early stages of the design, which can dramatically affect the energy consumption of the embedded system. Results from this analysis show that, the selection of the enhancement that has to be introduced in the IMO has to be based on correct decisions in the trade-offs that exist between energy budget, required performance, and area cost of the embedded system. This Chapter also shows different Pareto-optimal trade-off points in this design space.

4.2. Related Work and Motivating Example

As shown in Section 3.2, the CELB (*Central Loop Buffer Architecture for Single Processor Organisation*) represents the most traditional use of the

loop buffer concept. For more details see Section 2.3.2. Figure 4.1 shows the generic architecture of this IMO. This loop buffer architecture neither has partitioning in the loop buffer architecture nor in the program memory, and its connections depend on a single centralised component. Therefore, efficient parallelism in the execution of an application cannot be achieved in this kind of architectures due to the lack of hardware resources.

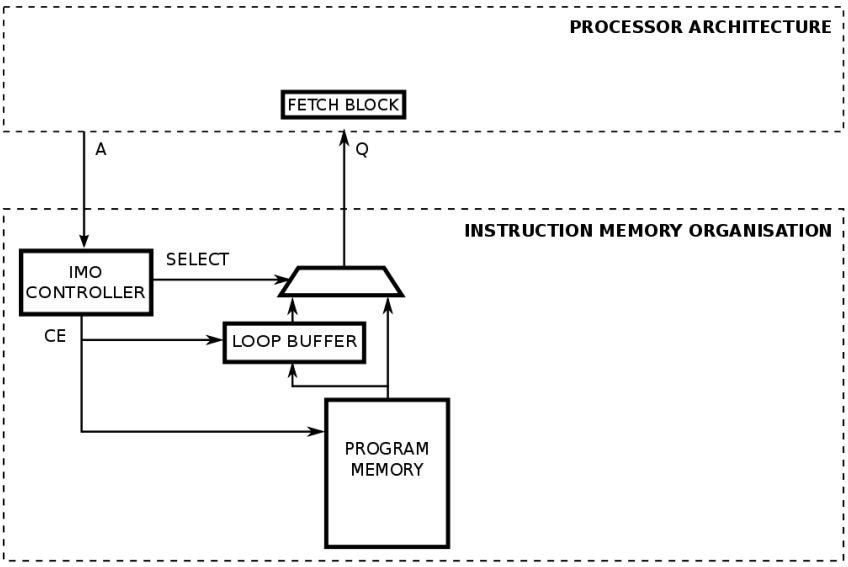


Figure 4.1: Instruction memory organisation with a central loop buffer architecture for single processor organisation.

The centralised resources and the global communication of single-threaded architectures make CELB architectures less energy efficient, when techniques of loop transformations are applied to exploit parallelism within loops. The CLLB (*Clustered Loop Buffer Architecture with Shared Loop-Nest Organisation*) mitigates these bottlenecks. For more details see Section 2.3.3. Figure 4.2 shows the generic loop buffer architecture of a CLLB architecture, which inner connections are controlled by one single component. In this loop buffer architecture, the loop buffer controller is more complex, because it controls the partitions that exist in the loop buffer architecture and in the program memory. This set of loop buffer architectures also includes all the enhancements that come from the introduction of low-power operating modes in the IMO, as well as from the research that was done in power management of banked memories [BMP00, FEL01, LK04].

Efficient parallelism exploitation is not yet fully achievable with CLLB architectures, due to the fact that loops with different threads of control have

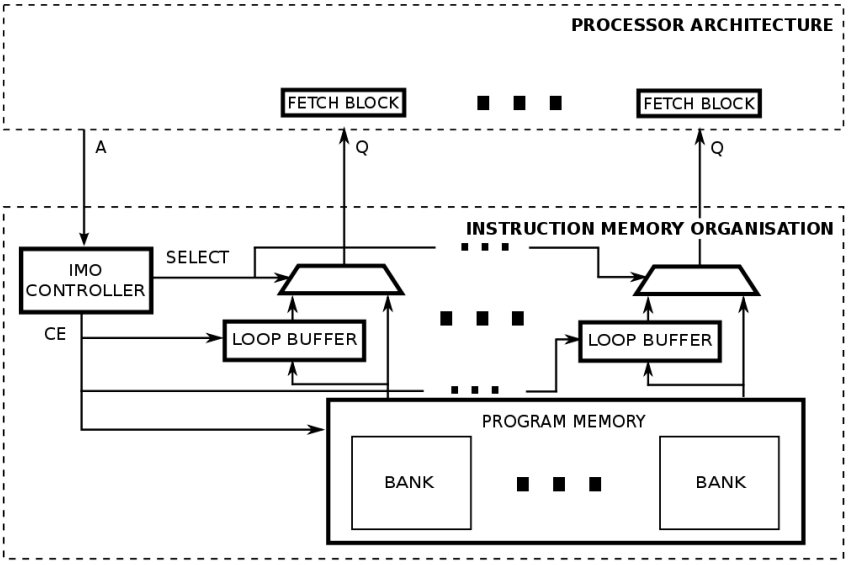


Figure 4.2: Instruction memory organisation with a clustered loop buffer architecture with shared loop-nest organisation.

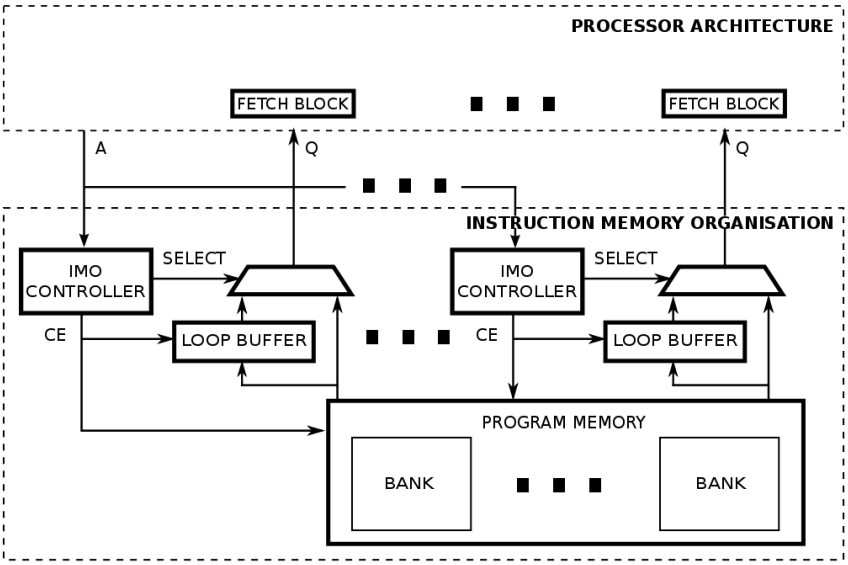


Figure 4.3: Instruction memory organisation with a distributed loop buffer architecture with incompatible loop-nest organisation.

to be merged into a single loop with a single thread of control. The required code transformation is performed using techniques like loop transformations (*e.g.*, loop fusion). However, not all loops of an application can be efficiently exploited by this manner. In the case of incompatible loops, the parallelism cannot be efficiently exploited because these loops require multiple loop controllers, which results in loss of energy and performance. Therefore a need exists for multi-threaded platforms that could support execution of multiple incompatible loops with minimal hardware overhead. That is achievable with the DLB (*Distributed Loop Buffer Architecture with Incompatible Loop-Nest Organisation*). In this loop buffer architecture not only the loop buffer memories are distributed across the IMO, but also the loop buffer controllers that manage them. Therefore, in this special set of loop buffer architectures, each loop buffer memory has its own local loop buffer controller. Due to this fact, DLB architectures can work like multi-threaded platforms, allowing the execution of incompatible loops in parallel with minimal hardware overhead. For more details see Section 2.3.4. Figure 4.3 shows the generic architecture of this recent loop buffer architecture. As shown in this Figure, the inner connections of this IMO are managed by a logic of controllers that is distributed across the architecture. In this IMO, partition exists in both the loop buffer architecture and the program memory. The controller of this IMO is even more complex than the controllers of the previous ones, because it also controls the execution of each loop in the corresponding loop buffer architecture to allow the parallel execution of loops with different iterators.

Table 4.1: Power consumption of loops that are executed over loop buffer memories with different sizes.

Loop Body Size [Instruction Word]	**Loop Buffer Size [Instruction Word]**		
	4	16	32
4	1.02×10^{-04} [W]	1.72×10^{-04} [W]	3.73×10^{-04} [W]
16	1.05×10^{-03} [W]	1.72×10^{-04} [W]	3.73×10^{-04} [W]
32	1.05×10^{-03} [W]	1.10×10^{-03} [W]	3.73×10^{-04} [W]

The high-level trade-off analysis of different loop buffer architectures that is presented in this Chapter differs from previous works like [BSBD+08] in the following points. First, this work depends only on the characteristics of the IMO and is not attached to any processor architecture. Second, it is not focused on a specific set of embedded systems as the benchmarks used in Section 4.4 prove. Third, the analysis proposes not only the architectural design, but also the implementation of the architecture. The purpose of the analysis presented in this Chapter can be clearly shown using the realistic illustrative example that is exposed in Figure 4.4, which presents the execution of a synthetic benchmark on the loop buffer architectures that are described in the previous paragraphs of this Section. Figure 4.4 is used to show the benefits and the drawbacks of each one of the loop buffer architectures. Assuming

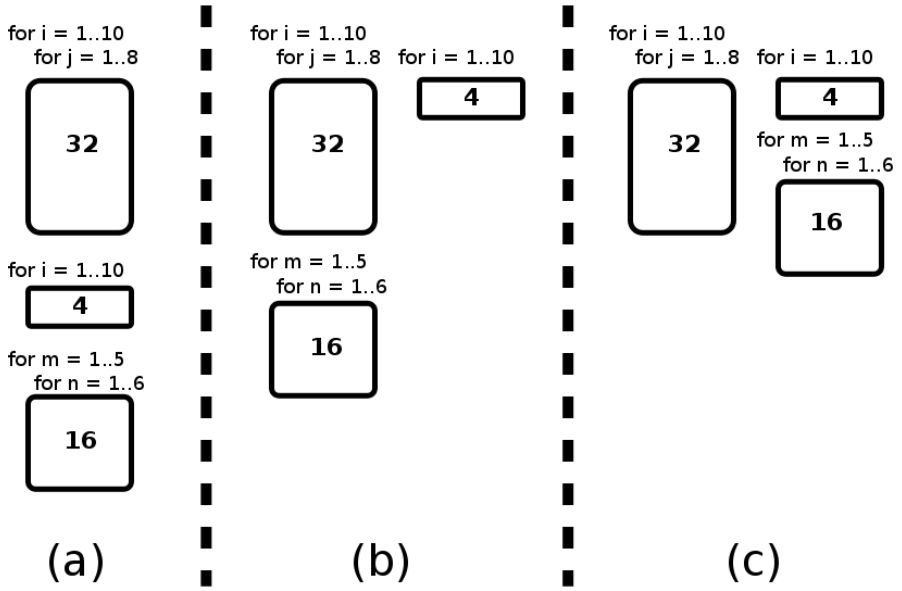


Figure 4.4: Execution of a synthetic benchmark in the representative loop buffer architectures.

that this benchmark is composed of three loops of 4, 16, and 32 instruction words respectively, its execution in a CELB architecture (see Figure 4.1) can be represented as shown in the case (a) of Figure 4.4. In this Figure, the loops are sequentially mapped during the execution of the benchmark. The loop buffer size is fixed to the size of the bigger loop from the set of loops that compose this benchmark. With this strategy, all loops that form the benchmark are stored without any split. If any split is present in the loops, part of the instructions are fetched from the program memory leading to reduce the energy savings of the loop buffer architecture. However, in this loop buffer architecture, no penalty exists in the performance of the benchmark, because jumps in the program memory are used to handle this situation. Using the values of power consumption that are presented in Table 4.1, the energy consumption of this loop buffer architecture is estimated for a specific system frequency of operation (*i.e.*, 100MHz) in Equation 4.1. Note that in all the calculations, E_{lbXLBY} is the energy that a loop of X instruction words of loop body consumes in a loop buffer architecture with a size of Y instruction words. It is possible to see that the calculation of this energy consumption E_{lbXLBY} is based on three components: the power consumption of the loop buffer architecture, the total number of accesses that are performed over the loop buffer architecture, and the time that requires one access in order to be performed. These three components are split by parenthesis in Equation 4.1, Equation 4.2, Equation 4.3, Equation 4.4, and Equation 4.5 in order to appreciate how they change between different cases.

$$\begin{aligned} E_{CELB} = & \quad E_{lb4LB32} + E_{lb16LB32} + E_{lb32LB32} \\ = & \quad ((3.73 \times 10^{-04}) \times (4 \times 10) \times (10^{-08})) \\ + & \quad ((3.73 \times 10^{-04}) \times (16 \times 30) \times (10^{-08})) \\ + & \quad ((3.73 \times 10^{-04}) \times (32 \times 80) \times (10^{-08})) = 1.15 \times 10^{-08} J \end{aligned} \tag{4.1}$$

If this benchmark is executed in a CLLB architecture (see Figure 4.2), case (b) of Figure 4.4, the first step is to analyse the data dependencies between loops and see which loops are incompatible. In this motivating example, because it is assumed that there are no data dependencies, only the loops that have a size of 4 and 32 instruction words can be executed in parallel. In this last case, the program memory and the loop buffer are split in smaller and individual sizes to fit the necessity of each instruction cluster that form the loop buffer architecture. The loop that has a size of 16 instruction words is incompatible with the other two loops, and this loop buffer architecture does not support execution of multiple incompatible loops in parallel. On the one hand, if two loop buffers with the same size are used (*i.e.*, 32 instruction words), the energy consumption is estimated by Equation 4.2. On the other hand, if the choice is to adapt the size of the loop buffers to the loops that are executed in them (*i.e.*, loop buffers of 4 and 32 instruction words), the energy consumption is estimated by Equation 4.3. Based on these results, it is possible to see that the improvement in energy savings, that comes from the use of CLLB architectures instead of CELB architectures, is related to the better adaptation of the sizes of the loop buffers to the sizes of the loops that form the application. Besides, in this case, NOP instructions are read from the banks that form the program memory, but not written to the loop buffers, decreasing the transactions between components of the IMO. This introduces a penalty on the performance of this benchmark in the CLLB architecture.

$$\begin{aligned} E_{CLLB1} = & \quad E_{lb4LB32} + E_{lb16LB32} + E_{lb32LB32} \\ = & \quad ((3.73 \times 10^{-04}) \times (4 \times 10) \times (10^{-08})) \\ + & \quad ((3.73 \times 10^{-04}) \times (16 \times 30) \times (10^{-08})) \\ + & \quad ((3.73 \times 10^{-04}) \times (32 \times 80) \times (10^{-08})) = 1.15 \times 10^{-08} J \end{aligned} \tag{4.2}$$

$$\begin{aligned} E_{CLLB2} = & \quad E_{lb4LB4} + E_{lb16LB32} + E_{lb32LB32} \\ = & \quad ((1.02 \times 10^{-04}) \times (4 \times 10) \times (10^{-08})) \\ + & \quad ((3.73 \times 10^{-04}) \times (16 \times 30) \times (10^{-08})) \\ + & \quad ((3.73 \times 10^{-04}) \times (32 \times 80) \times (10^{-08})) = 1.14 \times 10^{-08} J \end{aligned} \tag{4.3}$$

Finally, if the benchmark is executed in a DLB architecture (see Figure 4.3), the first step again is to analyse the data dependencies between loops, but in this case, there is no need to check whether the loops are not incompatible, due to the fact that this kind of loop buffer architecture supports execution of multiple incompatible loops in parallel. This scenario is shown in the case (c)

of Figure 4.4. Also in this last case, the program memory and the loop buffer are split in smaller and individual sizes to fit the necessity of each instruction cluster that form the loop buffer architecture. On the one hand, if two loop buffers with the same size are used (*i.e.*, 32 instruction words), the energy consumption is estimated by Equation 4.4. On the other hand, if the choice is to adapt the size of loop buffers to the loops that are executed over them (*i.e.*, loop buffers of 16 and 32 instruction words), the energy consumption is estimated by Equation 4.5. Based on these results, it is possible to conclude that any improvement in the ILP (*Instruction-Level Parallelism*) of the system brings improvements not only in performance, but also in the energy consumption of the system. The increase in ILP makes easy the adaptation of the sizes of the loop buffers to the sizes of the loops that form the application, because it gives more freedom to combine the execution of the loops that form the application. In this last loop buffer architecture, the control logic is more complex than the controllers of the previous loop buffer architectures, but this control logic considerable reduces the execution time and the overall energy of the application, because it allows the parallel execution of loops with different iterators.

$$\begin{aligned} E_{DLB1} = & \; E_{lb4LB32} + E_{lb16LB32} + E_{lb32LB32} \\ = & \; ((3.73 \times 10^{-04}) \times (4 \times 10) \times (10^{-08})) \\ + & \; ((3.73 \times 10^{-04}) \times (16 \times 30) \times (10^{-08})) \\ + & \; ((3.73 \times 10^{-04}) \times (32 \times 80) \times (10^{-08})) = 1.15 \times 10^{-08} J \end{aligned} \tag{4.4}$$

$$\begin{aligned} E_{DLB2} = & \; E_{lb4LB16} + E_{lb16LB16} + E_{lb32LB32} \\ = & \; ((1.72 \times 10^{-04}) \times (4 \times 10) \times (10^{-08})) \\ + & \; ((1.72 \times 10^{-04}) \times (16 \times 30) \times (10^{-08})) \\ + & \; ((3.73 \times 10^{-04}) \times (32 \times 80) \times (10^{-08})) = 1.04 \times 10^{-08} J \end{aligned} \tag{4.5}$$

As seen after comparing the loop buffer architectures, embedded systems designers can face a trade-off between the total energy budget of the system and the performance that is required by the application running on the embedded system. As shown in the motivating example of this Section, conventional loop buffer architectures (*i.e.*, CELB architectures and CLLB architectures) are not good enough in terms of energy efficiency due to the high overhead that these loop buffer architectures show. Due to this fact, DLB architectures appear as a promising option to improve the energy efficiency of IMOs. Section 4.4 not only presents the systematic analysis of this trade-off, but also clearly demonstrates that the area overhead and the selected memory technology for the loop buffer architecture have to be taken into account in this trade-off.

4.3. Experimental Framework

For the case study and the benchmarks that are presented in this Chapter, an experimental framework is built, which is composed of a DMH (*Data Memory Hierarchy*), an IMO, a processor architecture, a loop buffer architecture, and an I/O interface. The processor architectures are designed using *Target Compiler Technologies* [TAR12]. For more information see Appendix B. The ISA (*Instruction Set Architecture*) of these processor architectures are composed of integer arithmetic, bitwise logical, compare, shift, control, and indirect addressing I/O instructions. Apart from support for interrupts and on-chip debugging, these processor architectures support zero-overhead looping control hardware, which allows fast looping over a block of instructions. The energy simulations, that are shown in this Chapter, are performed using the high-level energy estimation and exploration tool that is described in Chapter 3. For a given application and compiler, this tool explores different loop buffer architectures and configurations that can compose the IMO. As shown in Section 3.3, in order to correctly model the loop buffer architectures that are presented in Section 4.2, this tool uses energy models of each one of the components that form the IMO. Based on these models, the characteristics of these components are adapted to the requirements of the processor architecture and the application that is executed in the embedded system. As shown in Section 3.3, this tool requires three inputs: the store access history report of the application, the requirements of the embedded system, and the cycle-level energy models of the memory instances that can compose the IMO. As outcome of the processing of its inputs, this high-level energy estimation and exploration tool provides the energy profiles of the components that form each one of the representative IMOs.

The loop buffer architecture consists of a loop buffer memory and a loop buffer controller. The implementation of the loop buffer memory is based on a set of banks, in which each bank can be configured to fit the desired size and number of instruction words. The loop buffer controller is the component that monitors the state of the loop buffer memory inside of the IMO. The energy models of the loop buffer memory and the loop buffer controller can be obtained in two ways. Either from the data sheets of the commercial memories that the designer wants to use (*e.g.*, a SRAM-based memory, register-based memory, FF-based memory), or by creating the energy models of the memory instances from post-layout back annotated energy simulations (*e.g.*, loop buffer controller, FFs). In this last option, RTL (*Register-Transfer Level*) simulations have to be performed to produce accurate energy waveforms of the instance under modelling with some user-specified training VCD (*Value Change Dump*) files. In order to cover the whole design space of an instance, variations in depth, width, and technology that is used in the implementation of the memory are done. The evaluation that is presented in this Chapter is

performed using TSMC (*Taiwan Semiconductor Manufacturing Company*) 90nm LP (*Low Power*) libraries and commercial memories.

4.4. Experimental Results

Real-life embedded applications are used as benchmarks to perform the high-level analysis proposed in this Chapter. Table 4.2 shows the characteristics of these real-life benchmarks. The selected benchmarks are prime examples not only of all application domains that are loop dominated, exhibit sufficient opportunity for DLP (*Data-Level Parallelism*) and/or ILP, comprise signals with multiple word-lengths, and require a relatively limited number of variable multiplications, but also of more general-purpose applications domains that can be found in the area of wireless base-band signal processing, multimedia signal processing, or different types of sensor signal processing. This selection of the benchmarks was performed with the goal of making the analysis that is presented in this Chapter generic enough, in order to be applicable to all loop-dominated application domains in embedded systems.

4.4.1. Energy Variation Influenced by Handling Conditions

When the IMO has no loop buffer architecture and it is based only in a single SPM (*Scratchpad Memory*), every instruction has to be stored in the program memory. Due to the fact that the size of each memory has to be power of two, the size of the program memory has to be the minimum number of rows that can contain the program code of the benchmark, being each row the number of bits of the instructions that form the program code of the benchmark. Besides, without a loop buffer architecture in the IMO, the complete instruction row has to be read from the program memory, even the NOP instructions. In this case, all the instruction clusters must have the same condition structure. This condition structure is normally based on two options: predication or jump in program memory. On the one hand, if it is based on predication, the true and the false branch are executed in parallel. On the other hand, if it is based on jump in program memory, jumps in program memory have to be done to the appropriate addresses. Therefore, predication increases the energy consumption and the execution time of the embedded system in comparison with jump in program memory. Figure 4.5 shows the energy consumption of these conventional architectures as reference compared to CELB architectures and CLLB architectures.

If the IMO uses a CELB architecture, the total storage size of the IMO is increased up to 25 % compared to the IMO based on a SPM architecture

Table 4.2: Profiling information of the benchmarks that are used in the design space exploration of the loop buffer schemes.

Benchmark [Reference]	Cycles	Issue Slots	Bits per Instruction	LB Size [Instructions]	Loop Code [%]	NOP Instructions [%]
AES [TSH+10]						
(See Section C.3)	3,347	1	16	32	77.44	0.09
BIO-IMAGING [PFH+12]						
(See Section C.4)	334,071	4	80	64	98.01	26.70
CWT NON-OPTIMISED [YKH+09]						
(See Section C.5)	274,464	1	16	32	6.61	0.48
CWT OPTIMISED [YKH+09]						
(See Section C.6)	102,827	1	20	64	6.36	2.50
DWT [DS98]						
(See Section C.7)	758,216	2	16	518	65.70	25.19
MRFA [QJ04]						
(See Section C.9)	177,170	2	32	64	19.13	17.01

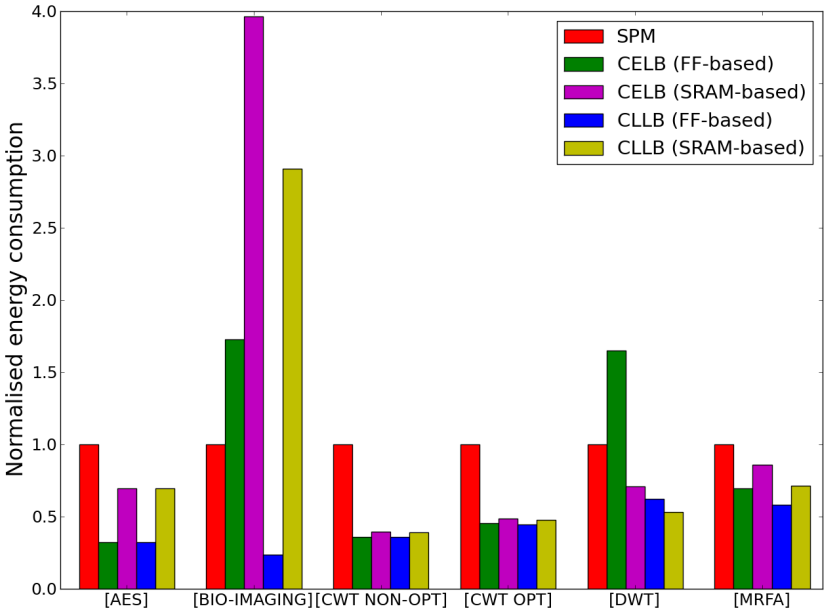


Figure 4.5: Normalised energy consumption in different IMOs running the selected benchmarks.

due to the introduction of the loop buffer architecture. The size of the loop buffer architecture is the size of the memory that can contain the bigger loop of the set of loops that form the program code of the benchmark. It should be noted that this memory should preferably be a power of two for practical implementations. In this case, the total energy consumption of the components that form the IMO is bigger than the IMO without loop buffer architecture due to the increase of the components and connections that form this IMO. However, as it is possible to see from Figure 4.5, the increase of the number of accesses to the components inside of the IMO does not increase the total energy consumption of the IMO. Indeed, this total energy consumption is lower (up to 67%) because the largest part of the total number of accesses that are done in the IMO are focused on the loop buffer architecture. Based on Table 4.2, it is possible to see that the expected energy reduction depends not only on the total percentage of time that the execution is done in loop code, but also on the size of the loop buffer that is used. It is clear that the number of loop iterations has to be taken into account to evaluate when it makes sense to introduce a loop buffer. The benchmarks *BIO-IMAGING* and *DWT* are exceptions to this rule, as explained in Section 4.4.2. The CELB architecture has the same conditional structure as a IMO based on a single SPM. However, in this case, jumps in loop buffer architecture can be

supported or not. If jumps in loop buffer architecture are not supported, the IMO increases its energy consumption, because the loop buffer architecture is not used due to the jump, and the read access has to be done in the program memory. Therefore, predication and jumps in program memory are the options that strongly increase the energy consumption of the embedded system in comparison with jumps in loop buffer architecture.

If the IMO includes a CLLB architecture, the program memory and the loop buffer architecture are split into smaller and individual banks to fit the necessity of each instruction cluster that form the architecture. In this case, the total storage size of the IMO is not modified compared to the case of the CELB architecture, because equal instructions within a loop need to be stored multiple times. Due to the fact that in this IMO NOP instructions are read from the banks that form the program memory but not written to the loop buffer architectures, the number of instructions that have to be read and write inside of the IMO decreases. As shown in Figure 4.5, the CLLB architecture decreases (42 % in average) the total energy consumption of the IMO compared to the CELB architecture, and this decrease directly depends on the number of NOP instructions that are contained in the loop code. Besides, in this case, jumps in loop buffer architectures are supported by default. However, in this case, there are more options in how the conditional program code is handled. If different conditions exist across all the loop buffer architectures, it is necessary to make these conditions compatible across all the loop buffer architectures. In order to solve this problem, more NOP instructions have to be present in the loop buffer architectures, but these are not read. The impact of handling conditions is well appreciated in the benchmarks *BIO-IMAGING* and *DWT*, where the handling conditions used by the CLLB architecture improves considerably the energy consumption of the IMO compared to the handling conditions used by the CELB architecture.

4.4.2. Energy Variation Influenced by Technology

In Section 4.4.1, it is visible that when the IMO uses a CELB architecture, in the case of the benchmarks *BIO-IMAGING* and *DWT*, the increase of the number of accesses to the components that are inside of the IMO increases the total energy consumption of the IMO. In order to analyse this fact, Figure 4.5 presents a FF-based implementation and a SRAM-based implementation of the CELB architecture. If the energy consumption of these two implementations are compared throughout all the benchmarks that are used in this Chapter, it is possible to see that the FF-based implementation is more energy efficient than the SRAM-based implementation for applications with limited storage requirements. However, due to the bigger sizes of the loop

buffer architecture, in the benchmarks *BIO-IMAGING* and *DWT*, SRAM-based implementations are better options. Then, the storage implementation technology plays an important role. And whenever feasible, the required size should be reduced by applying appropriate code transformations and compiler optimisations on the code that has to be stored in the IMO and especially in the loop buffer architecture.

Figure 4.6, based on area-energy Pareto curves, shows the overhead in the memory area that is required in order to achieve the energy savings of two of the benchmarks of Table 4.2. As shown in this Figure, the area penalty, that the embedded systems designer has to assume in order to achieve further energy savings, is relatively small (10 % in average). But this still shows an interesting trade-off that has to be decided based on the overall design context. The application characteristics (*i.e.*, the number of NOP instructions in loops) have to be analysed in order to make the decision in this trade-off. If the application has a high percentage of NOP instructions in loops, the CLLB architecture has to be selected. Otherwise, the CELB architecture has to be selected.

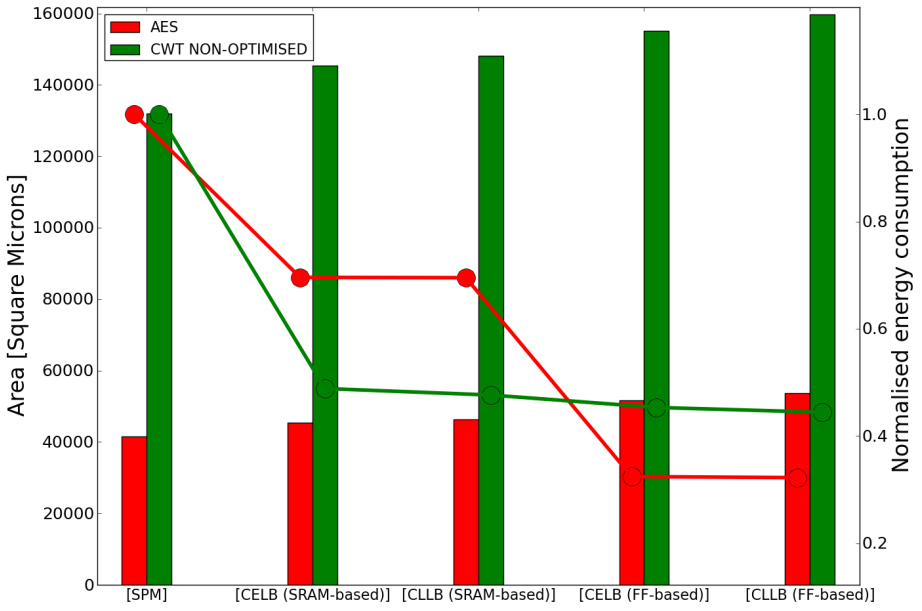


Figure 4.6: Normalised energy consumption (represented by lines) vs. occupancy (represented by solid bars) in different IMOs.

4.4.3. Energy Variation Influenced by Code Transformations and Compiler

Figure 4.7 contains a bar plot where the DLB architecture is compared to the CELB architecture and the CLLB architecture. In this Figure, it is possible to see how the code transformations and the optimisations of the compiler affect the energy consumption of the IMO. The baseline architecture is again an IMO that is based only on a SPM. As shown in Figure 4.7, on the one hand, the difference in energy consumption between the CELB architecture and the CLLB architecture is related to the loop transformations that are applied in the application in order to parallelise its execution. An effective parallelism leads to higher difference between these two architectures, where the CLLB architecture has less energy consumption. However, a poor parallelism will lead to small difference between them, and even, in some cases the CLLB architecture could have more energy consumption than the CELB architecture. On the other hand, the difference in energy consumption between the CLLB architecture and the DLB architecture is related to the behaviour of the compiler. If the compiler maps the application on the architecture effectively, the difference between energy reductions will increase, where the DLB architecture will be the loop buffer architecture that has less energy consumption. These results show that an energy efficient embedded design has to take care not only about the architectural configurations that are used in the embedded system, but also about the characteristics of the application that is running on it.

4.4.4. Discussion and Summary of the Pareto-Optimal Trade-Offs for Embedded Systems Designers

From the experimental results that are obtained along this Section, it is possible to conclude that each loop buffer architecture is clearly optimal for a specific pattern of application code and a specific set of requirements of the embedded system. Due to this fact, a high-level energy analysis has to be performed by embedded systems designers in the first steps of the design process of an embedded system to not only increase the energy savings, but also have a better distribution of the energy budget throughout the whole embedded system.

The author of this Ph.D. thesis proposes that the high-level energy analysis has to be performed based on the following guidelines:

- **Use of a loop buffer architecture in the IMO.** In order to introduce a loop buffer architecture not only the total percentage of time from the execution of the application that is related to loop codes has to be taken into account, but also the size and the width of the

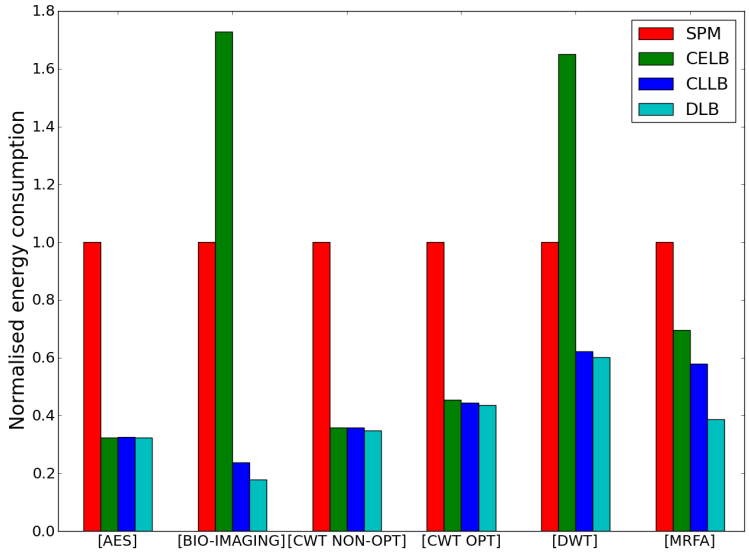


Figure 4.7: Normalise energy consumption of CELB architectures, CLLB architectures, and DLB architectures that are used with the real-life benchmarks.

loop buffer memories that are introduced in the IMO. As shown in Section 4.4.2, the FF-based implementation is more energy efficient than the SRAM-based implementation for applications with limited storage requirements. However, for applications that require big sizes of loop buffer memories, SRAM-based implementations are better options. The energy savings that are obtained from the introduction of the loop buffer architecture have to compensate the energy consumption related to the increase of the components and the connections that form the IMO.

- **Use of CELB architectures.** The use of CELB architectures in the IMO is related to the parallelism that can be exploited in the embedded architecture. If there is no parallelism or a poor parallelism exists, CELB architectures are the best option to use. Indeed, the difference that can exist among the energy consumption of the CELB and the rest of loop buffer architectures is directly related to the loop transformations that can be applied in the application in order to obtain an effective parallelism in its execution.

- **Use of CLLB architectures.** The use of CLLB architectures in the IMO depends not only on the parallelism that can be exploited in the application code, but also on the number of NOP instructions that are contained in the loop code. If the application presents a high number of

NOP instructions and a high-efficient parallelism can be achieve on it, this loop buffer architecture is better option than CELB architectures.

- **Use of DLB architectures.** The use of DLB architectures in the IMO depends not only on the parallelism and the number of NOP instructions that are contained in the loop code of the application, but also on the behaviour of the compiler. If the compiler can map the application code on the embedded architecture effectively, the DLB architecture will be the loop buffer architecture that has less energy consumption.

In the case that the design of an embedded system takes into account not only the reduction of the energy consumption of the IMO, but also the area occupancy and the penalty of performance that can appear when a loop buffer architecture is introduced, the following trade-offs have to be considered:

- **Area occupancy vs. energy consumption.** As it is possible to see from Figure 4.8, any architectural enhancement that is introduced in the IMO produces an increase in the area occupancy. From this Figure, it is also possible to see that when the complexity of the loop buffer architecture is increased in terms of number of components and interconnections, the area occupancy of the embedded system is also increased. However, this Figure clearly shows that the more complex architectures, the potentially much more energy-efficient are.
- **Performance penalty vs. energy consumption.** Figure 4.9 shows that there is not a penalty in the execution time of an application when a loop buffer architecture is introduced in the IMO. Indeed, as shown in Figure 4.9, the increase in complexity of the implementation of the loop buffer architecture produces a reduction in execution time. This is directly related to the high-efficient parallelism that the complex loop buffer architectures can achieve. Besides, in this trade-off, the condition structure that is implemented in the processor architecture has to be also taken into account. As it was shown in Section 4.4.1, predication and jumps in program memory are the options that strongly increase the energy consumption and the execution time of the embedded system in comparison with jumps in loop buffer architecture.

Based on previous guidelines, the optimal energy efficient IMO for a given application code and embedded architecture has to be selected among the complementary implementations of loop buffer architectures that cover the distinct partitions of the design space of the loop buffer architecture. In order to find this optimal energy-efficient IMO, interesting trade-offs have to be decided based on the overall design context, which is mainly based on the patters and the characteristics of the application code and the processor architecture that form the embedded system.

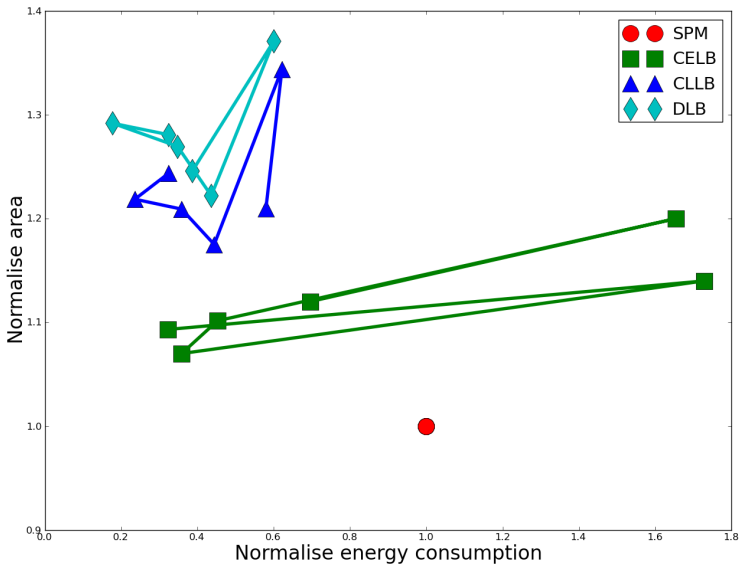


Figure 4.8: Normalised energy consumption vs. area occupancy of the real-life benchmarks on the different representative IMOs.

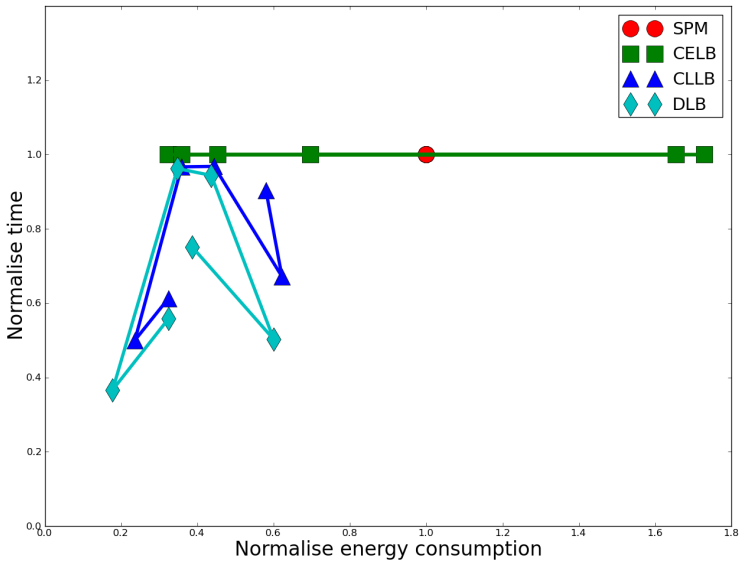


Figure 4.9: Normalised energy consumption vs. performance penalty of the real-life benchmarks on the different representative IMOs.

4.5. Example of Implementation of a Loop Buffer Architecture for Power Optimisation of Dynamic Workload Applications

In this Section, a run-time self-tuning banked loop buffer architecture for power optimisation of dynamic workload applications is presented. The design and the implementation of this special loop buffer architecture is showed as a realistic illustrative example of how embedded systems designers have to use the guidelines that are described in Section 4.4.4. These guidelines are used to implement an efficient loop buffer architecture for a given application code and embedded architecture. In this approach, a run-time loop buffer controller is used to steer the banked loop buffer architecture in order to optimise both the dynamic and the leakage energy consumption of the IMO. The decisions of the loop buffer controller are based on the actual input loop, and a predictor that accurately predicts the future state of the loop buffer memory considering the last states. Results show that using banking in a loop buffer architecture leads to higher reduction in the total energy consumption of the IMO if the tuning approach is applied sparingly. Based on post-layout simulations, the loop buffer controller that is presented in this Section improves the total energy consumption by average of 20 % in comparison with a loop buffer architecture based on a single monolithic memory, and more than 90 % in comparison with IMOs without loop buffer architectures. The approach that is proposed in this Section differs from the related work that is presented in Section 1.3 in the following points. Firstly, this approach uses a run-time controller to steer the banked loop buffer architecture. Secondly, this approach combines the optimisation of both the dynamic and the leakage energy.

4.5.1. Design of the Loop Buffer Architecture

4.5.1.1. System Overview

Figure 4.10 shows the experimental framework. The system is composed of a DMH, an IMO, a processor architecture, a loop buffer architecture, and an I/O interface. In the next paragraphs, the processor architecture and the loop buffer architecture are described in detail.

The processor architecture of the system is designed using *Target Compiler Technologies* [TAR12]. For more information see Appendix B. The ISA of this processor architecture is composed of integer arithmetic, bitwise logical, compare, shift, control, and indirect addressing I/O instructions. Apart from support for interrupts and on-chip debugging, this processor architecture supports zero-overhead looping control hardware. This feature allows fast looping over a block of instructions. Once the loop is set using a special

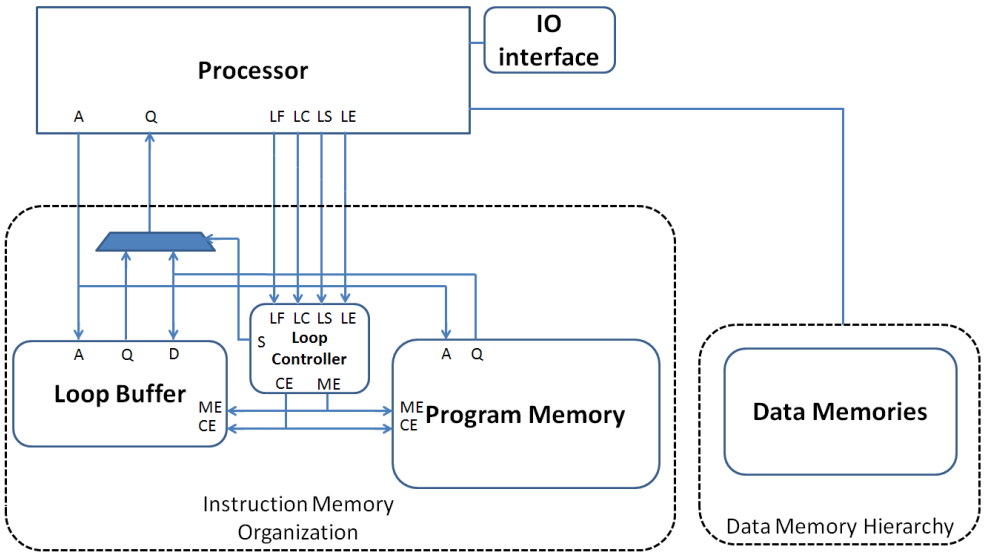


Figure 4.10: Experimental framework for the run-time self-tuning banked loop buffer architecture for power optimisation of dynamic workload applications.

instruction, additional instructions are not needed in order to control the execution of the loop. This is due to the fact that the loop is executed a pre-specified number of iterations (known at compile time). The status of this dedicated hardware is stored in the following registers:

LS **Loop Start address register.** This register stores the address of the first instruction of the loop.

LE **Loop End address register.** This register stores the address of the last instruction of the loop.

LC **Loop Count register.** This register stores the remaining number of iterations of the loop.

LF **Loop Flag register.** This register keeps track of the hardware loop activity. Its value represents the number of nested loops that are active.

Figure 4.10 shows that the loop buffer architecture consists of a loop buffer memory and a loop buffer controller. The implementation of the loop buffer memory is based on a set of banks, in which each bank is a FF array that can be configured to fit the desired size and number of instruction words. The choice of a FF-based implementation is due to the energy reduction of using FF arrays instead of SRAM-based memories for small memory sizes [VM07]. The loop buffer controller is the component that monitors the state of the loop buffer memory. The zero-overhead looping control hardware provides to the

loop buffer controller run-time information of each executed loop. Based on this information, the loop buffer controller decides the state that minimises the energy consumption of the IMO. Constraints on the maximum number and size of banks that form the loop buffer memory need to be also enforced in the optimisation process. The states are predicted using a model that takes into account the last H states. Run-time profiling registers are used to store them. This model is described in detail in Section 4.5.1.2.

4.5.1.2. System Operation

In essence, the operation of the loop buffer concept is as follows. During the first iteration of the loop, the instructions are fetched from the program memory to both the loop buffer architecture and the processor architecture. In this iteration, the loop buffer architecture records the instructions of the loop. Once the loop is stored, for the rest of iterations, the instructions are fetched from the loop buffer architecture instead of the program memory. In the last iteration, the connection between the processor architecture and the program memory is restored, such that subsequent instructions are fetched from the program memory. During the execution of non-loop parts of the application code, instructions are fetched directly from the program memory.

The loop buffer controller monitors this operation based on the state-machine that is shown in Figure 4.11. The six states of the state-machine are:

- **s0** Initial state.
- **s1** Transition state between *s0* and *s2*.
- **s2** State where the loop buffer architecture is recording the instructions that the program memory supplies to the processor architecture.
- **s3** Transition state between *s2* and *s4*.
- **s4** State where the loop buffer architecture is providing the instructions to the processor architecture.
- **s5** Transition state between *s4* and *s0*.

The transition states *s1*, *s3*, and *s5* are necessary in order to give the control of the instruction supply from the program memory to the loop buffer architecture and vice-versa. The transition between *s4* and *s1* is necessary because the body size of a loop can change in real-time (*i.e.*, a loop body with if-statements or function calls). In order to check in real-time whether the loop body size changes or not, a 1-bit tag is used.

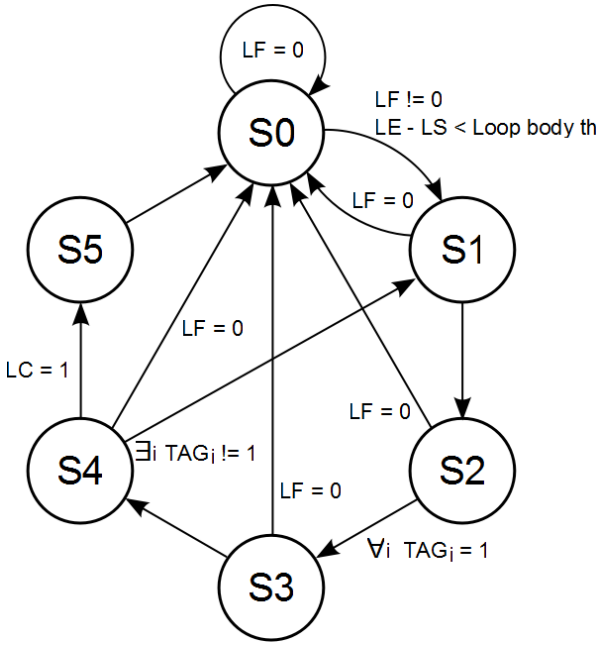


Figure 4.11: State-machine diagram of the loop buffer controller.

In order to provide an IMO based on a banked loop buffer architecture, the operation of this approach is performed in two phases:

- **Off-line phase.** The loop buffer memory is designed in this phase. The leakage and the dynamic energy consumption of each element that forms the loop buffer architecture are calculated based on the technology parameters that are used in the design. These data and the user-defined parameters that are used in the optimisation process are specified to create a Pareto curve that relates the size and the number of memory banks versus the energy access per instruction. The most energy efficient configuration of loop buffer memory is selected from this curve. In architectures with large number of banks, this algorithm can have prohibitive cost for embedded systems due to their limited computational capabilities. Therefore, the parameters of the predictor are obtained at design time by testing the predictor using execution characteristics that are derived from profiled benchmarks.

- **Run-time phase.** The loop buffer controller, which is composed of the state-machine and the predictor, is implemented as dedicated logic hardware. The predictor, which is made using a heuristic, generates the future state of the loop buffer memory considering the probability that the last H states are repeated. The inputs for the predictor are obtained via control wires from the locations in the rest of the platform where relevant information is available. Based on the input information

that are supplied by the zero-overhead looping control hardware and the predictor, the loop buffer controller finds the optimum state of the loop buffer memory in order to minimise the energy consumption of the IMO. This target is accomplished by minimising a cost function under the specific constraints that have been derived from the off-line phase. This cost function is described in detail in Section 4.5.1.3.

4.5.1.3. Problem Formulation

Loop Model

The information of the input loop is composed of two parameters: size of the loop and its number of iterations. The size of the loop is defined as a vector $L_t \epsilon \Re$, where L_t is the loop body size of the loop that is starting at time t. The number of iterations of the loop is defined as a vector $I_t \epsilon \Re$, where I_t is the number of iterations of the loop executed at time t. In this model, both vectors are assumed to be continuous and ranging from zero to a max value as described formally by Equation 4.6 and Equation 4.7.

$$0 \leq L_t \leq L_{max} \quad \forall t \tag{4.6}$$

$$0 \leq I_t \leq I_{max} < \infty \quad \forall t \tag{4.7}$$

On the one hand, the value L_{max} is limited by the size of the loop buffer memory. Loops with higher size than the size of the loop buffer memory are partially loaded in the loop buffer architecture filling it completely. On the other hand, the value I_{max} is given based on the assumption that the execution of the loop finishes at a certain moment.

$$0 \leq S_t \leq B_{max} \quad \forall t \tag{4.8}$$

After the off-line phase, the configuration of the loop buffer architecture is assumed to be fixed. Due to this constraint, a limited set of splits can be chosen for a specific loop. A split of the input loop is defined as a vector $S_t \epsilon \Re^n$, where $(S_t)_i$ is the number of instruction words that are stored in bank i at time t, and n is the number of banks that form the loop buffer memory. Equation 4.8 describes formally the range of sizes that a split can have. The size of each split is assumed to be continuous and range from zero to the value of the size of the biggest bank (B_{max}) of the loop buffer architecture.

Loop Buffer Architecture Model

In this model, the size of each bank that forms the loop buffer memory is defined as a vector $B_i \epsilon \Re$, where B_i is the size of the bank i. The sizes of the banks are assumed to be continuous and range from zero to a maximum size value B_{max} as it is described in Equation 4.9. The value B_{max} as well as the size of each bank are defined in the off-line phase. These sizes are selected to keep the addressing logic as simple as possible.

$$0 \leq B_i \leq B_{max} \quad \forall i \tag{4.9}$$

$$A_t = S_t \iff (S_t)_i \leq B_i \quad \forall i \tag{4.10}$$

The accesses are defined as a vector $A_t \epsilon \Re^n$, where $(A_t)_i$ is the number of accesses to the bank i at time t, and n is the number of banks that form the loop buffer memory. The relation between the accesses to a memory bank $(A_t)_i$ and the possible splits of the input loop $(S_t)_i$ is expressed by Equation 4.10. From this Equation, it is possible to derive that $A \subseteq S$.

Energy Evaluation Model

The total energy consumption of every bank that forms the loop buffer memory is defined as a vector $E_i \epsilon \Re$, which is modelled as summation of the dynamic and the static energy consumption as shown in Equation 4.11. On the one hand, the dynamic energy consumption $(E_{dynamic})_i$ is directly proportional to the number of accesses to B_i due to the execution of a specific L_t. On the other hand, the static or leakage energy consumption $(E_{leakage})_i$ of B_i is composed of the energy that is consumed in the operating modes that are used in the execution of a specific L_t. Dynamic and leakage energy consumption are described by Equation 4.12, Equation 4.13, and Equations 4.14 respectively.

$$E_i = (E_{dynamic})_i + (E_{leakage})_i \tag{4.11}$$

$$(E_{dynamic})_i = (E_{access})_i (A_t)_i I_t + (E_{access})_{PM} (2L_t) \tag{4.12}$$

$$(E_{leakage})_i = ((E_{act})_i + (E_{off})_i + (E_{ret})_i) I_t \tag{4.13}$$

The total energy consumption E_i of the next state that is assigned to B_i is computed based on Equation 4.11. Matrix $(E_{access})_i \in \Re^n$ contains the dynamic energy consumption of each one of the banks that form the loop buffer memory, whereas matrix $(E_{access})_{PM} \in \Re$ contains the dynamic energy consumption of the program memory. Matrices $((C_t)_i)_{act} \in \Re^n$, $((C_t)_i)_{off} \in \Re^n$, and $((C_t)_i)_{ret} \in \Re^n$ contain the leakage energy consumption of B_i working in *active*, *off*, and *retention* operating mode respectively.

$$(E_{act})_i = ((C_t)_i)_{act}(A_t)_i \tag{4.14a}$$
$$(E_{off})_i = ((C_t)_i)_{off}(L_t(B_i - sign((A_t)_i))) \tag{4.14b}$$
$$(E_{ret})_i = ((C_t)_i)_{ret}(sign((A_t)_i)L_t - (A_t)_i B_i) \tag{4.14c}$$

Minimisation Objective Function

The target is described as an energy minimisation problem of a linear time-discrete system that is subjected to constraints. The assignment problem of the state of the loop buffer memory is solved by minimising function Δ:

$$minimise \quad \Delta = \alpha|E_{sel} - E_{loop}| + (1 - \alpha)|E_{sel} - E_{sys}| \tag{4.15}$$

Function Δ is expressed by a weighted sum of two terms, where E_{sys} is the energy consumption of the state of the loop buffer memory that is proposed by the predictor, E_{loop} is the energy consumption of the state of the loop buffer memory that is optimum for a given L_t, and E_{sel} is the energy consumption of the state of the loop buffer memory for a given A_t. The energy consumption overhead for starting up a change in the state of the loop buffer memory is employed for each bank. This energy can be defined as a vector $(E_t)_i \epsilon \Re$, where $(E_t)_i$ is the transition energy of the bank i due to the transitions between operating modes. Taking into account this overhead, the selected energy consumption of the state of the loop buffer memory E_{sel} is formally described by Equation 4.16:

$$E_{sel} = E_i + (E_t)_i \tag{4.16}$$

Weighting scalars are included in the summation to increase the accuracy of the prediction. The weighting scalar α is the probability to repeat the execution of L_t, which is obtained using the Poisson distribution over the run-time profiling registers.

Every time the minimisation is performed, the next state of the loop buffer memory is computed. In order to create a feedback loop, the solution is updated with the recent state of the loop buffer memory in the run-time profiling registers. This formulation performs a state of the loop buffer memory assignment by embedding the prediction of the future state into the optimisation process. The weighting scalar α can be modified to achieve a good prediction after testing the predictor on benchmarks.

4.5.2. Experimental Results of the Design of the Loop Buffer Architecture

The evaluation of the proposed approach is performed using four real-life embedded applications for nodes of biomedical WSN (*Wireless Sensor Network*): two algorithms of heartbeat detection (see Section C.5 and Section C.6), and two versions of the cryptographic algorithm AES (*Advanced Encryption Standard*) (see Section C.2 and Section C.3). *B1* and *B2* correspond with the heartbeat detection algorithms, while *B3* and *B4* correspond with the cryptographic algorithms. Based on the profiles of these benchmarks, the configuration of the IMO that the off-line phase suggests is 16-bit/1KB SRAMs as program memory and data memory respectively, and a loop buffer memory of 63 instruction words with the sizes $B = [32, 16, 8, 4, 2, 1]$. To evaluate the energy impact, post-layout simulations are used to have an accurate estimation of parasitics and switching activity. The evaluation is performed using TSMC 90nm LP libraries and commercial memories [VIR12]. A clock frequency of 100MHz is selected. Although a specific configuration is analysed in this Section, this experimental framework is generic enough that can be easily modify and extended in the off-line phase to target other benchmarks.

4.5.2.1. Run-Time Execution Behaviour

The run-time behaviour of the proposed heuristic is shown in Figure 4.12. In order to represent the states of the loop buffer memory, vector $sign(A_t)$ is expressed as a decimal value. For instance, if only the banks with size eight and two are active, the activation vector that is based on the loop buffer configuration of this experimental evaluation is $sign(A_t) = [0, 0, 1, 0, 1, 0]$, which can be represented by the decimal value 10. In this Figure, three states of the loop buffer memory can be seen at the same time: the state that is proposed by the predictor, the state that is optimum for a given loop, and the decision made by the loop buffer controller. As shown in this Figure, our approach decides to activate the states of the loop buffer memory that are optimum for the loops when their execution time is large. However, if the input

loop has not a large execution time, our approach gives more importance to the state suggested by the predictor and, based on the possible energy transitions, decides the most energy efficient state for the IMO. Therefore, it is possible to conclude that the loop buffer controller plays an optimum role of switching on the banks in terms of energy efficiency.

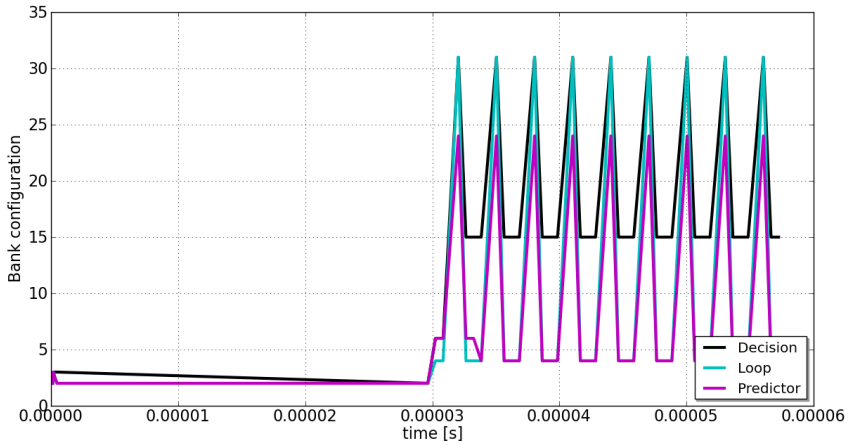


Figure 4.12: Run-time execution behaviour of the heuristic in benchmark *B1*.

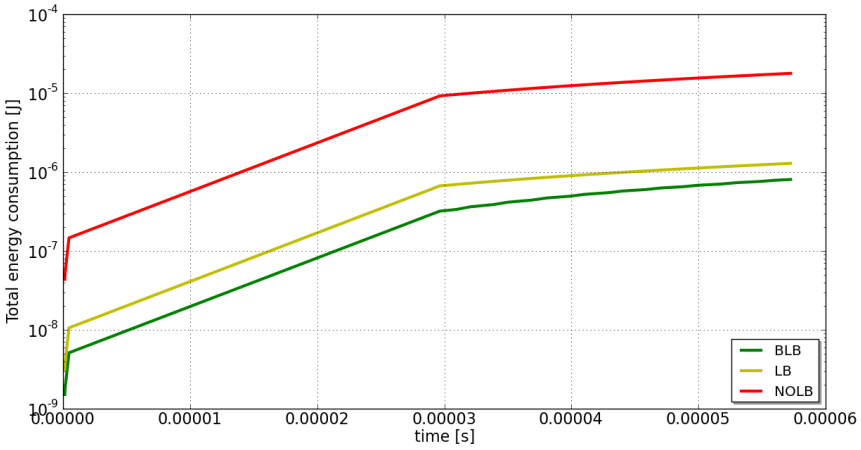


Figure 4.13: Total energy consumption of the IMO implementations in benchmark *B1*.

Keeping the analysis on the simulation of the same benchmark, Figure 4.13 depicts the total energy consumption of three different implementations of the IMO for the same benchmark: an IMO with NOLB (*NO Loop Buffer*)

architecture, an IMO with a LB (*Loop Buffer*) architecture based on a single monolithic memory, and an IMO with a BLB (*Banked Loop Buffer*) architecture. In this Figure, the total energy consumption is the summation of the dynamic and the leakage energy consumption. As can be seen, the most energy efficient implementation is the BLB architecture. The difference in energy consumption between the implementations that are based on the loop buffer architectures cannot be appreciated correctly, because in Figure 4.13 the total energy consumption is represented in logarithm scale. To overcome this handicap, Table 4.3 presents the total energy consumption of the execution of every benchmark on these three implementations of the IMO. From this Table, it is possible to see how this method improves the energy consumption in an average of 20 % in comparison with the LB architecture, and more than 90 % in comparison with IMOs without loop buffer architectures.

Table 4.3: Total energy consumption of the implementations of the IMO.

	B1	**B2**	**B3**	**B4**
NOLB	54.4250 [μJ]	8.89640 [μJ]	796.860 [μJ]	417.500 [μJ]
LB	3.9345 [μJ]	0.64314 [μJ]	57.606 [μJ]	30.186 [μJ]
BLB	3.6859 [μJ]	0.49461 [μJ]	31.477 [μJ]	25.031 [μJ]

In order to extend the energy analysis between the loop buffer architectures, Figure 4.14 and Figure 4.15 present the dynamic energy consumption and the leakage energy of both loop buffer architectures. On the one hand, the analysis of the dynamic energy consumption shows that, there are specific patterns of loop arrivals that make the dynamic energy consumed by the BLB architecture higher than the dynamic energy consumed by the LB architecture. As shown in Figure 4.12 and Figure 4.14, it is possible to see that the higher energy consumption of the BLB architecture is due to the first big change in the state of the loop buffer memory. However, after this learning step, it is possible to see how the predictor helps the loop buffer controller to avoid this extra dynamic energy consumption, leading to further dynamic energy savings than the LB architecture. On the other hand, the leakage energy consumption of the BLB architecture is always lower than the leakage energy consumption of the LB architecture. As for dynamic energy consumption, it is possible to see from Figure 4.15 that for specific patterns of loop arrivals, the leakage energy consumed by the BLB architecture is close to the energy consumed by the LB architecture, but it is still lower. With this result, it is possible to conclude that the off-line phase of this approach did a good selection of banks for the loop buffer memory configuration. From both energy analyses, the conclusion is that the leakage energy consumption is dominant in these architectures. Besides, the actual gain or loss trade-off fully depends on the run-time situation. Therefore, an online-controller is essential to exploit this.

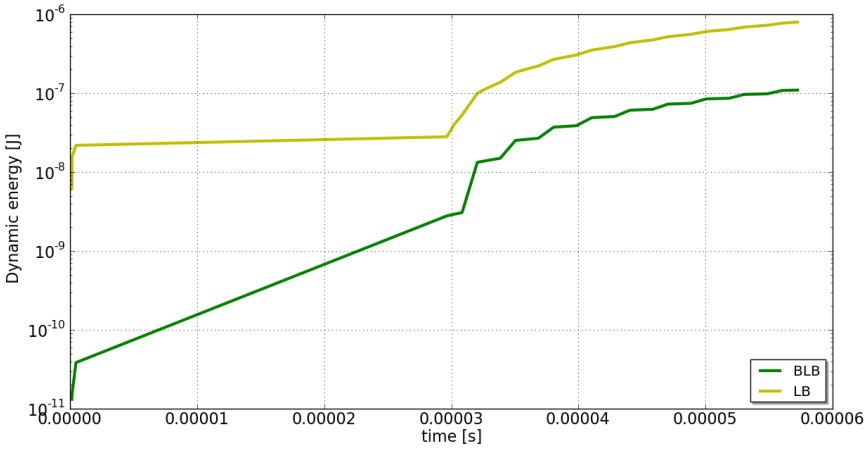


Figure 4.14: Dynamic energy consumption of the loop buffer architectures in benchmark *B1*.

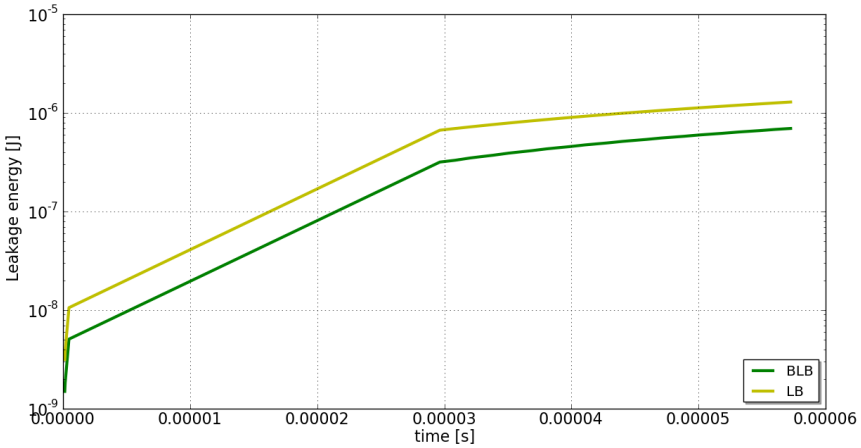


Figure 4.15: Leakage energy consumption of the loop buffer architectures in benchmark *B1*.

4.5.2.2. Energy Variation Influenced by H

The behaviour of the heuristic is influenced by the last states H that are stored in the run-time profiling registers. Table 4.4 shows the variation in the total energy consumption of the banked loop buffer architecture, based on the number of last states that are used by the predictor. As shown in Table 4.4,

the longer the number of register are used, the less total energy reduction can be achieved. However, a small number of registers is only needed for the prediction, what allows to this approach to keep the control logic as simple as possible. Due this fact, with only an increase of 5 %–10 % of the area of the IMO due to the introduction of the loop buffer controller, the energy overhead for running the loop buffer controller is compensated greatly with the energy savings of using small memory instances. The value of the previous increase depends on the instruction width and the size of the memory instances that have to be controlled.

Table 4.4: Total energy consumption using different H.

H	**B1**	**B2**	**B3**	**B4**
1	3.7418 [μJ]	7.95780 [μJ]	36.9494 [μJ]	25.4209 [μJ]
2	3.7017 [μJ]	7.95780 [μJ]	36.9494 [μJ]	25.3884 [μJ]
3	3.7338 [μJ]	0.49461 [μJ]	36.9494 [μJ]	25.4206 [μJ]
4	3.7017 [μJ]	0.49461 [μJ]	31.4770 [μJ]	25.3881 [μJ]
5	3.6859 [μJ]	0.49461 [μJ]	31.4770 [μJ]	25.0310 [μJ]
6	3.6780 [μJ]	0.49461 [μJ]	31.4770 [μJ]	25.0310 [μJ]
7	3.6780 [μJ]	0.49461 [μJ]	31.4770 [μJ]	25.0310 [μJ]
8	3.6780 [μJ]	0.49461 [μJ]	31.4770 [μJ]	25.0310 [μJ]

4.6. Conclusion

A relevant portion of the total energy budget of an embedded platform is related to the instruction memory organisation. Therefore, any enhancement that is introduced in this component of the system becomes crucial in order to decrease this energy bottleneck. This Chapter presented a high-level trade-off analysis of existing loop buffer schemes that helps embedded systems designers to make the correct decision in the trade-off that exists between the energy budget of the system and the performance and the area occupancy that are required. This Chapter shows that the possible energy reductions (up to 76 %) depends not only on the total percentage of time that the execution is done in loop code, but also on the size and the implementation of the loop buffer architecture that is used. Besides, handling conditions can make a really bad impact in the total energy consumption depending on the loop buffer architecture that is used. In this Section, a self-tuning banked loop buffer architecture was also proposed. A run-time loop buffer controller was used to steer the banked loop buffer architecture in order to combine the optimisation of both the dynamic and the leakage energy consumption of the IMO. The decisions of the loop buffer controller were based on the actual input loop, and a predictor that accurately predicted the future state of the loop buffer

architecture considering the last states. Results from the implementation of this special loop buffer architecture showed that the leakage energy was dominant in these architectures. Besides, the actual gain or loss trade-off fully depended on the run-time situation. Therefore, an online-controller became essential to exploit this. Based on post-layout simulations, this approach improves the energy consumption of the IMO by average of 20 % in comparison with a loop buffer architecture based on a single monolithic memory, and more than 90 % in comparison with IMOs without loop buffer architectures.

In the next chapter...

the reader will find the results that are obtained from the application of the loop buffer concept in real-life embedded applications that are widely used in the nodes of a biomedical WSN (*Wireless Sensor Network*). These results will show which scheme of loop buffer is more suitable for applications with certain behaviour.

Chapter 5

Case Study of Power Impact of Loop Buffer Schemes for Biomedical Wireless Sensor Nodes

"Study not to know more, but to know better."

— Lucius Annaeus Seneca.

In this Chapter, the loop buffer concept is applied in real-life embedded applications that are widely used in the nodes of biomedical wireless sensor networks, to show which scheme of loop buffer is more suitable for applications with certain behaviour. Post-layout simulations demonstrate that a trade-off exists between the complexity of the loop buffer architecture and the energy savings of utilising it. Therefore, the use of loop buffer architectures in order to optimise the instruction memory organisation from the energy efficiency point of view should be evaluated carefully, taking into account two factors. First, the percentage of the execution time of the application that is related to the execution of loops. Second, the distribution of the execution time percentage over each one of the loops that form the application.

5.1. Introduction

As shown in Section 1.1.1, embedded systems have different characteristics compared with general-purpose systems. On the one hand, embedded systems combine software and hardware to run a specific and fixed set of applications. However, these applications differ greatly in their characteristics, because they range from multimedia consumer devices to industry control systems. On the

other hand, unlike general-purpose systems, embedded systems have restricted resources and a low-energy budget. In addition to these restrictions, embedded systems often have to provide high computation capability, meet real-time constraints, and satisfy varied, tight, and time conflicting constraints in order to make themselves reliable and predictable. Section 1.2 shows that the combination of these requirements and the fact that the well-known problem of the *memory wall* becomes even greater in embedded systems make the decrease of the total energy consumption of the system become a big challenge for designers, who have to consider not only the performance of the system, but also its energy consumption. Works like [HP07, VM07, CRL+10] demonstrate that the IMO (*Instruction Memory Organisation*) and the DMH (*Data Memory Hierarchy*) take portions of chip area and energy consumption that are not negligible. In fact, both memory architectures now account for nearly 40 %–60 % of the total energy budget of an embedded instruction-set processor platform (see Figure 1.7). Therefore, the optimisation in energy consumption of both memory architectures becomes extremely important.

J. Villarreal *et al.* [VLCV01] showed that 77 % of the execution time of an application is spent in loops of 32 instructions or less. This demonstrated that in applications of signal and image processing, a significant amount of the total execution time was spent in small program segments. If these could be stored in smaller memory banks (*e.g.*, in the form of a LB (*Loop Buffer*)), the dynamic energy consumption could be reduced significantly. The energy-saving features of the loop buffer concept can be obtained in Figure 5.1, where it is shown that accesses in a small memory have lower energy consumption than in a large memory. This is the base of the loop buffer concept, which is a scheme to reduce the dynamic energy consumption in the IMO. Furthermore, banking is identified as an effective method to reduce the leakage energy consumption in memories [KKK02]. Apart from the possibility of using low-power operating modes, the use of memory banks reduces the effective capacitance as compared with a single monolithic memory.

Embedded systems constitute the digital domain of the nodes of a WSN (*Wireless Sensor Network*). They are widely deployed in several types of systems ranging from industrial monitoring to medical applications. Particularly, for the biomedical domain, the information that is processed and transmitted is confidential or requires authentication in the majority of the cases. Due to this fact, it is not unusual that two applications like a HBD (*Heartbeat Detection*) algorithm and a cryptographic algorithm such as AES (*Advanced Encryption Standard*) can be found in the nodes of biomedical WSNs. These two real-life embedded applications are used in this Chapter as case studies to evaluate the energy reductions achieved by the use of the IMOs that are based on the loop buffer concept.

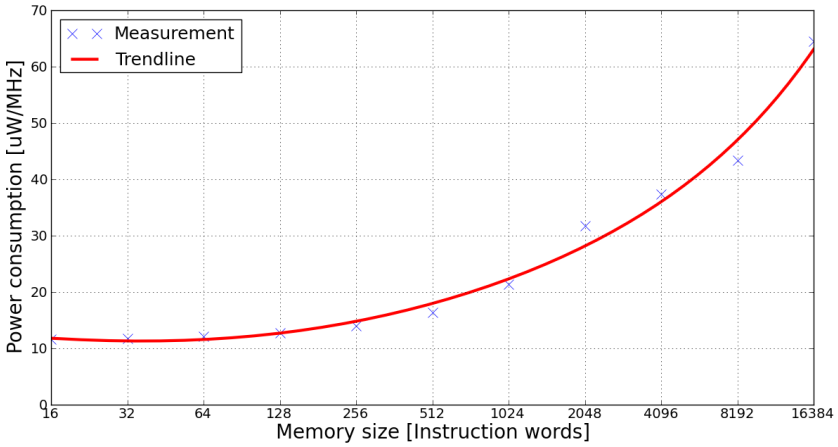


Figure 5.1: Power consumption per access in 16-bit word SRAM-based memories designed by *Virage Logic Corporation* tools [VIR12] using TSMC 90nm LP process.

In this Chapter, the loop buffer concept is applied in the two real-life embedded applications that are described in the previous paragraph. The loop buffer architectures that are analysed in this Chapter are the CELB (*Central Loop Buffer Architecture for Single Processor Organisation*) and the BCLB (*Banked Central Loop Buffer Architecture*). The contributions of this Chapter include:

- An analysis of real-life embedded applications that is used to show which type of loop buffer scheme is more suitable for applications with certain behaviour.
- The use of post-layout simulations to evaluate the energy impact of the loop buffer architectures, in order to have an accurate estimation of parasitics and switching activity.
- Gate-level simulations demonstrate that a trade-off exists between the complexity of the loop buffer architecture and the power benefits of utilising it. The use of loop buffer architectures in order to optimise the IMO from the energy efficiency point of view should be evaluated carefully. Two factors have to be taken into account in order to implement an energy efficient IMO based on a loop buffer architecture:
  - the percentage of the execution time of the application that is related to the execution of the loops included in the application.
  - the distribution of the execution time percentage, which is related to the execution of the loops, over each one of the loops that form the application.

5.2. Related Work

Researchers have demonstrated that the IMO can contribute to a large percentage of the total energy consumption of the embedded system (*e.g.*, [CRL+10]). Most of the architectural enhancements that have been used to reduce the energy consumption of the IMO have made use of the loop buffer concept. Works [Zha05, IMM02, BHK+97, LMA99, KGMS97, VSB04, TGN02, BHPS99] present the most traditional use of the loop buffer concept: the CELB (*Central Loop Buffer Architecture for Single Processor Organisation*). Work [Zha05] proposed a configurable instruction cache, which could be tailored for a particular application in order to utilise the sets efficiently, without any increase in cache size, associativity, or access time. Work [IMM02] proposed an alternative approach to detect and remove unnecessary tag-checks at run-time. Using execution footprints, which were recorded previously in a BTB (*Branch Target Buffer*), it was possible to omit the tag-checks for all instructions in a fetched block. If loops could be identified, fetched, and decoded only once, work [BHK+97] proposed an architectural enhancement that could switch off the fetch and the decode logic. The instructions of the loop were decoded and stored locally, from where they were executed. The energy savings came from the reduction in memory accesses as well as the lesser use of the decode logic. In order to avoid any performance degradation, work [LMA99] implemented a small instruction buffer that was based on the definition, the detection and the utilisation of special branch instructions. This architectural enhancement had neither an address tag store nor a valid bit associated with each loop cache entry. Work [KGMS97] evaluated the Filter Cache. This enhancement was an unusually small first-level cache that sacrificed a portion of performance in order to save energy. The program memory was only required when a miss occurs in the Filter Cache, otherwise it remained in standby mode. Based on this special loop buffer enhancement, work [VSB04] presented an architectural enhancement that detected the opportunity to use the Filter Cache, and enabled or disabled it dynamically. Also, work [TGN02] introduced a DFC (*Decoder Filter Cache*) in the IMO to provide directly decoded instructions to the processor architecture, reducing the use of the instruction fetch and decode logic. Furthermore, work [BHPS99] proposed a scheme, where the compiler generated code in order to reduce the possibility of a miss in the loop buffer cache. However, the drawback of this work was the trade-off between the performance degradation and the power savings, which was created by the selection of the basic blocks. For more details see Section 2.3.2.

Parallelism is a well-known solution for increasing performance efficiency. Due to the fact that loops form the most important part of an application [VLCV01], loop transformation techniques are applied to exploit parallelism within loops on single-threaded architectures. Centralised resources and global

communication make these architectures less energy efficient. In order to reduce these bottlenecks, several solutions that used multiple loop buffers were proposed in literature. Works [ZFMS05, ZLM07, BSBD+08] are examples of the work done in this field: the CLLB (*Clustered Loop Buffer Architecture with Shared Loop-Nest Organisation*). On the one hand, work [ZFMS05] presented a distributed control-path architecture for DVLIW (*Distributed Very Long Instruction Word*) processors, which overcame the scalability problem of VLIW (*Very Long Instruction Word*) control-paths. The main idea was to distribute the fetch and the decode logic in the same way that the register file was distributed in a multi-cluster data-path. On the other hand, work [ZLM07] proposed a multi-core architecture that extended traditional multi-core systems in two ways. First, it provided a dual-mode scalar operand network to enable efficient inter-core communication without using the memory. Second, it could organise the cores for execution in either coupled or decoupled mode through the compiler. These two modes created a trade-off between communication latency and flexibility, which should be optimised depending on the required parallelism. Work [BSBD+08] analysed a set of loop buffer architectures for efficient delivery of VLIW instructions, where the evaluation included the cost of the memory accesses and the wires that were necessary in order to distribute the instruction bits. For more details see Section 2.3.3.

The CELB architecture and the CLLB architecture can be implemented based on memory banks or without them. Power management of banked memories has been investigated from different angles including hardware, OS (*Operating System*) and compiler. Using memory access patterns in embedded systems, L. Benini *et al.* [BMP00] proposed an algorithm to partition on-chip SRAM (*Static Random Access Memory*) into multi-banks that could be accessed independently. X. Fan *et al.* [FEL01] presented memory controller policies for memory architectures with low-power operating modes. C. Lyuh *et al.* [LK04] used a compiler directed approach to determine the operating modes of memory banks after scheduling the memory operations. As it is possible to see from previous approaches, the drawback of using multiple buffers is usually the increase of the logic that controls the banks, which has the benefit of further decreasing the leakage energy consumption. This fact leads to the increase of the interconnect capacitances, as well as the reduction of the possible dynamic energy savings that are related to the access to smaller memories. Most approaches that are related to caches assume that the automated tuning is done statically, meaning that the tuning is done once during application design time. A. Ghosh *et al.* [GG04] presented a heuristic that, through an analytical model, directly determined the configuration of the cache based on the designer's performance constraints and application characteristics. Other cache tuning approaches could be used dynamically, while the application was executed [GRVD09].

5.3. Experimental Framework

This Section describes all the components that compose the experimental framework. On the one hand, Section 5.3.1, Section 5.3.2, and Section 5.3.3 describe the processor architectures that are used in this Chapter. On the other hand, Section 5.3.4 presents the rest of the components that compose the experimental framework and explains how the experimental framework is built based on a platform that can contain any processor architecture.

5.3.1. General-Purpose Processor

The general-purpose processor architecture is designed using the tools from *Target Compiler Technologies* [TAR12]. For more information see Appendix B. The ISA (*Instruction Set Architecture*) of this processor architecture is composed of integer arithmetic, bitwise logical, compare, shift, control, and indirect addressing I/O instructions. Apart from support for interrupts and on-chip debugging, this processor architecture supports zero-overhead looping control hardware, which allows fast looping over a block of instructions. Once the loop is set using a special instruction, additional instructions are not needed in order to control the loop, because the loop is executed a pre-specified number of iterations (known at compile time). This loop buffer implementation supports branches, and in cases where the compiler cannot derive the loop count, it is possible to inform the compiler through source code annotations that the corresponding loop will be executed at least N times, and at most M times, such that no initial test is needed to check whether the loop has to be skipped. The status of this dedicated hardware is stored in the following set of special registers:

- LS **Loop Start address register.** This register stores the address of the first instruction of the loop.
- LE **Loop End address register.** This register stores the address of the last instruction of the loop.
- LC **Loop Count register.** This register stores the remaining number of iterations of the loop.
- LF **Loop Flag register.** This register keeps track of the hardware loop activity. Its value represents the number of nested loops that are active.

The experimental framework uses an I/O interface in order to provide the capability of receiving and sending data in real-time. This interface is implemented in the processor architecture by FIFO (*First In, First Out*) architectures that are directly connected to the register file. The data memory that is required by this processor architecture in order to be a general-purpose processor is a memory with a capacity of 16K words/16 bits, whereas the required program memory is a memory with a capacity of 2K words/16 bits.

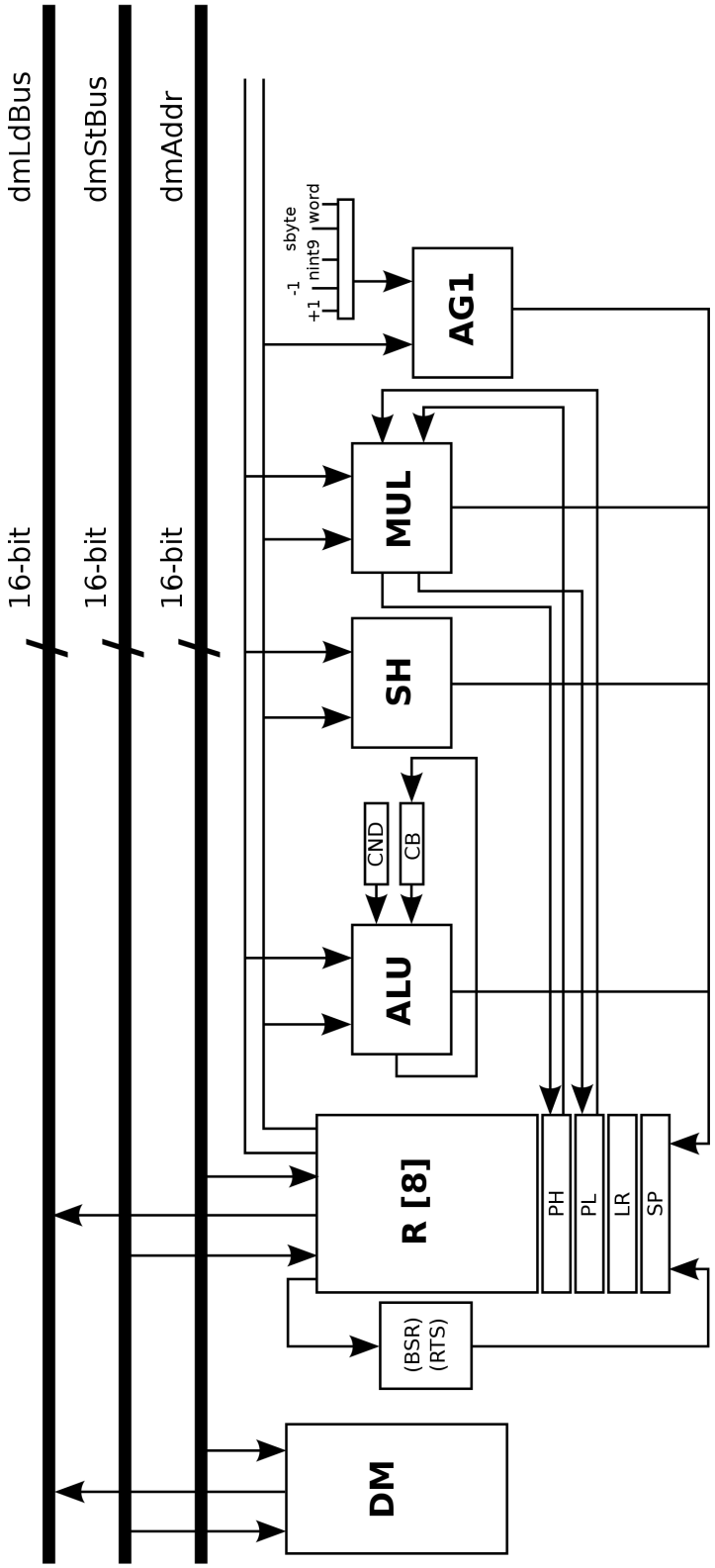


Figure 5.2: Data-path of the general-purpose processor.

Figure 5.2 presents the data-path of this processor architecture, where the main blocks are DM (*Data Memory*), R (*Register File*), ALU (*Arithmetic Logic Unit*), SH (*Shift Unit*), MUL (*Multiplication Unit*), and AG (*Address Generation Unit*). The address generation unit specifies the next address as a normal instruction word in the case of the *word* label, as negative offsets to the stack pointer register in the case of the *nint9* label, and as a relative offset of short jump instructions in the case of the *sbyte* label. In Figure 5.3, the main blocks are PM (*Program Memory*), PC (Program Counter), and the registers IR (ID) and IR (E1) which are related to the decode and the execute stage of the processor pipeline.

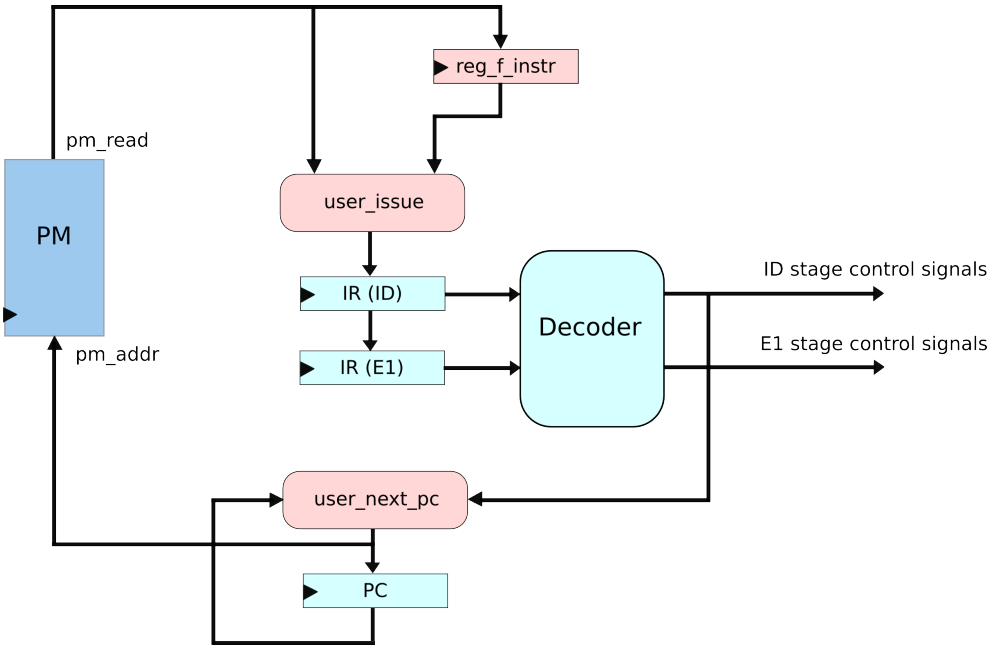


Figure 5.3: Control-path of the general-purpose processor.

5.3.2. Optimised Processor for the Heartbeat Detection Algorithm

The processor architecture that is optimised for the heartbeat detection algorithm (see Section C.6) is based on the processor architecture that is presented in Section 5.3.1. This Section presents the modifications and the optimisations that are performed in order to build this optimised processor architecture.

From the deep analysis that has to be performed to design the ASIP (*Application-Specific Instruction-Set Processor*) design for the heartbeat

detection algorithm, a loop is pointed out as the performance bottleneck in this specific algorithm. This loop performs the convolution operation, which is the core of the CWT (*Continuous Wavelet Transform*). A signed multiplication, whose result is accumulated in a temporally variable, is performed inside of this critical loop. The execution of this instruction is 72 % of the execution time of the algorithm according to profiling information. Therefore, in order to improve the performance, the MUL unit is modified to multiply two signed integers and accumulate, without shifting, the result of the multiplication. This optimisation saves energy and at the same time reduces both the complexity of the MUL unit and the execution time of the application.

The load operations that are related to the previous MUL operation are combined in a customised instruction in order to be executed in parallel. However, in the general-purpose processor, it is only possible to load and store data from the same memory once per stage of the pipeline. To solve this bottleneck, the main data memory is split in two identical data memories: DM (*Data Memory*) and CM (*Constant Memory*). In order to access two memories in parallel, another address generator (AG2) is created such that the load and the store operations from the DM and the CM can be performed at the same stage of the pipeline. As the input registers of the MUL unit can be loaded directly, a new modification can be performed. The parallel load and the MUL instruction are combined, by adding another stage in the pipeline and creating a custom instruction that integrates both instructions. The MUL instruction is then executed in the second stage of the pipeline, while the parallel load instruction is executed in the first stage of the pipeline. After this last modification, the MUL operation that is included in the main critical loop of this algorithm is performed using only one assembly instruction.

In a similar way to the MUL operation, another critical loop is optimised by combining load, select, and equal instructions in order to be executed in parallel. This instruction is created adding the functionality of the equal and the select instructions, and combining both of them with a normal load operation. The functional unit ALU 2 is created for this specific operation.

It should be noted that, apart from the specialised instructions that are described in previous paragraphs, custom techniques like source code transformations (*e.g.*, function combination, loop unrolling) and mapping optimisations (*e.g.*, look-up tables, elimination of divisions and multiplications, instruction set extensions) are applied to generate a more efficient code.

All the optimisations and the modifications that are described in this Section result in the new processor architecture shown in Figure 5.4. Basically, an

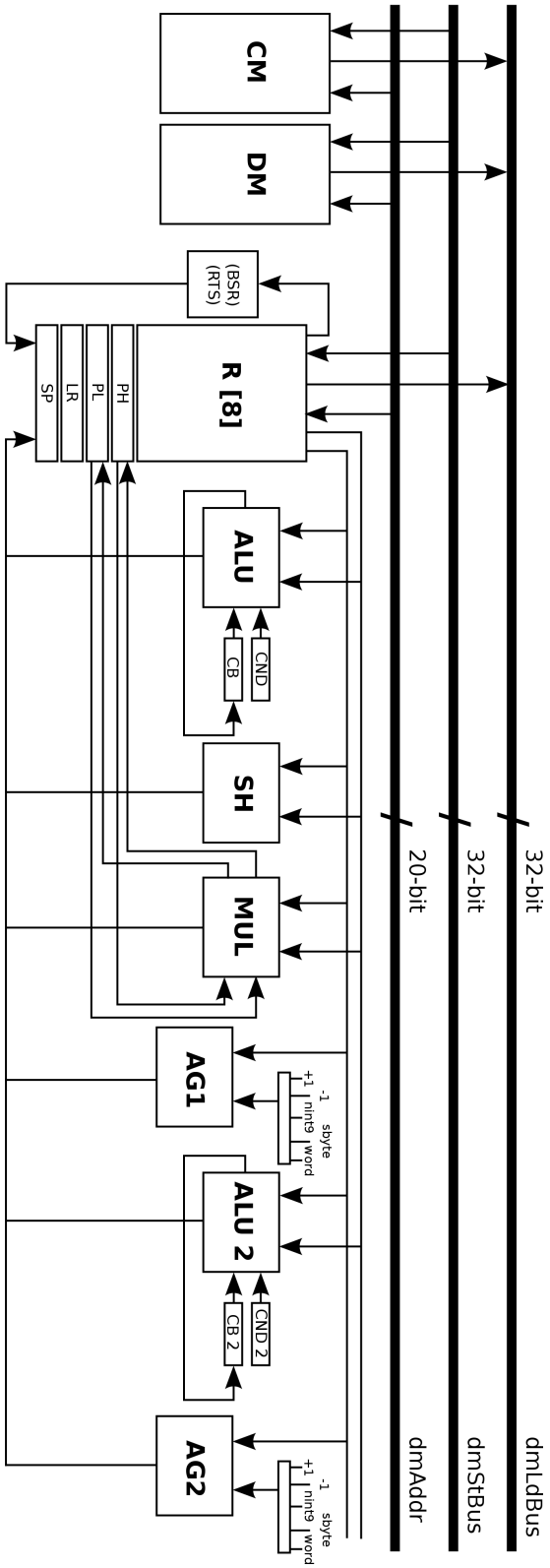


Figure 5.4: Data-path of the processor that is optimised for the heartbeat detection algorithm.

address generator (AG2) and a second ALU (ALU 2) are added, in addition to some pipes and ports. Apart from that, the program counter is modified to handle instructions that use 32-bit immediate values. In order to handle ECG (*Electrocardiogram*) signals sampled at 1KHz, the memories that are required by this processor architecture are a DM with a capacity of 8K words/32 bits, and a CM with a capacity of 8K words/32 bits. Besides, the program memory that is required by this processor architecture is a memory with a capacity of 1K words/20 bits. This optimised processor architecture is an implementation that is based on the work presented in Reference [YKH+09].

5.3.3. Optimised Processor for the AES Algorithm

The processor architecture that is optimised for the AES algorithm (see Section C.3) is based also on the processor architecture that is presented in Section 5.3.1. Analysing this algorithm, the critical functions are identified and optimised in order to improve performance in terms of clock cycles and memory accesses. Custom techniques like source code transformations (*e.g.*, function combination, loop unrolling) and mapping optimisations (*e.g.*, look-up tables, elimination of divisions and multiplications, instruction set extensions) are applied to generate a more efficient code.

In the design of this optimised processor, the structure of the general-purpose processor is kept intact (16-bit data-path), and an extra 128-bit data-path is added. This last data-path is connected with a VM (*Vector Memory*), a V (*Vector Register File*), and a Vector Unit (Functional Vector Unit). This unit includes the AES accelerating operations, as well as the logic and the arithmetic instructions that this algorithm requires. In this processor, the ISA is also extended with one AES accelerating instruction that has two inputs: a 128-bit input, which can be the state or a round key, and an integer input, which indicates the behaviour of the instruction itself. Depending on the input, the output contains the state or a round key. One of the advantages of this design is the ability to use the larger vector units only when they are required.

All the optimisations and the modifications that are presented in this Section result in the new processor architecture shown in Figure 5.5. Basically, an extra 128-bit data-path is added. This extra data-path includes a VM (*Vector Memory*), a V (*Vector Register File*), and a Vector Unit (Functional Vector Unit). In order to handle an input signal of 1,460 bytes, the DM required by this processor architecture is a memory with a capacity of 1K words/16 bits, and the VM is a memory with a capacity of 64 words/128 bits. Moreover, the required program memory is a memory with a capacity of 1K words/16 bits. This optimised processor architecture is an implementation that is based on the work presented in [TSH+10].

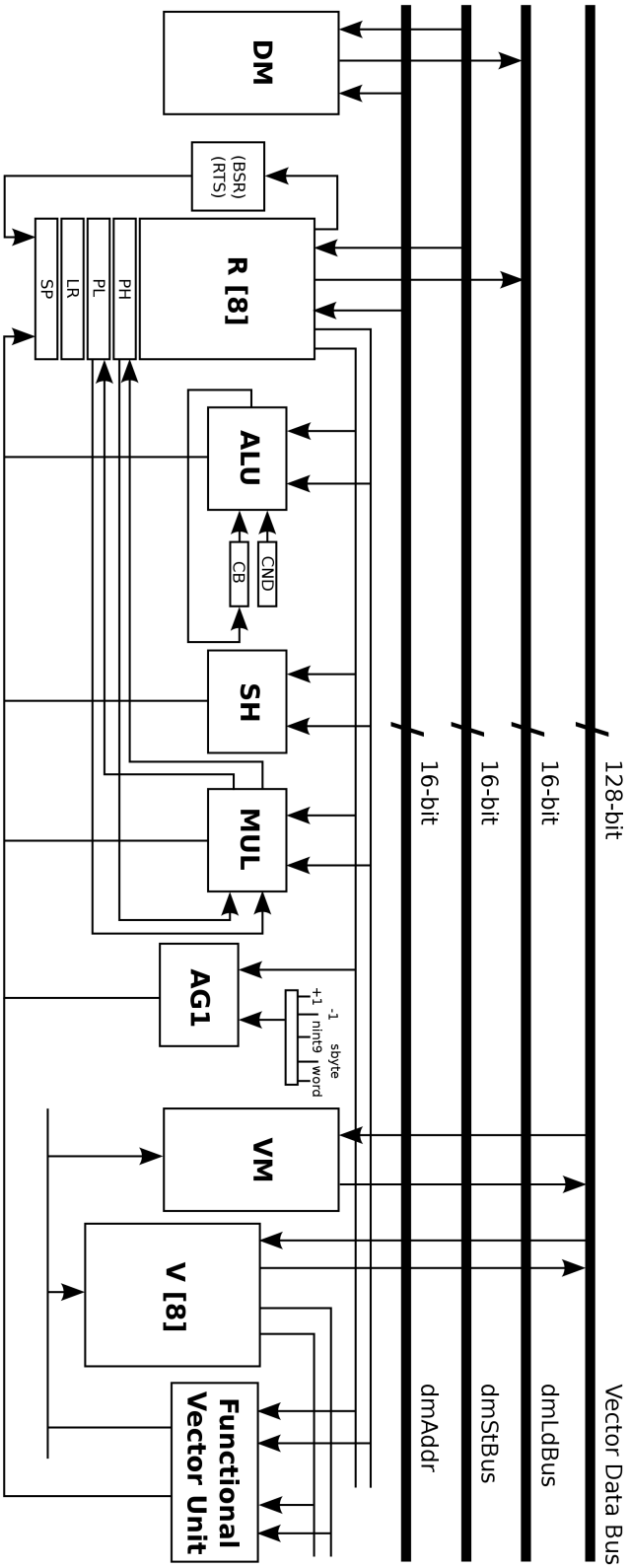


Figure 5.5: Data-path of the processor that is optimised for the AES algorithm.

5.3.4. Experimental Platform

The experimental platform is automatically generated for any of the processor architectures described in Section 5.3.1, Section 5.3.2, and Section 5.3.3. The experimental platform is composed of a DMH, an IMO, an I/O interface, and a processor architecture that is used as core of the embedded instruction-set processor platform. On the one hand, the program memory and the data memory are SRAM-based memories designed by *Virage Logic Corporation* tools [VIR12]. On the other hand, the I/O interface that provides the capability to receive and send data in real-time is connected with the I/O interface that is described in Section 5.3.1.

The interface between a processor architecture and an IMO is depicted in Figure 5.6. The interconnections of the processor architecture, the program memory, the loop buffer memory and the loop buffer controller are included in this Figure. Every component that forms the IMO is explained in the next paragraphs. In our experimental platform, the loop buffer architecture, which is composed of the loop buffer memory and the loop buffer controller, can be configured to be used as a CELB architecture or a BCLB architecture. For simplicity, the CELB architecture is used in the next paragraphs to explain the operation of the loop buffer concept.

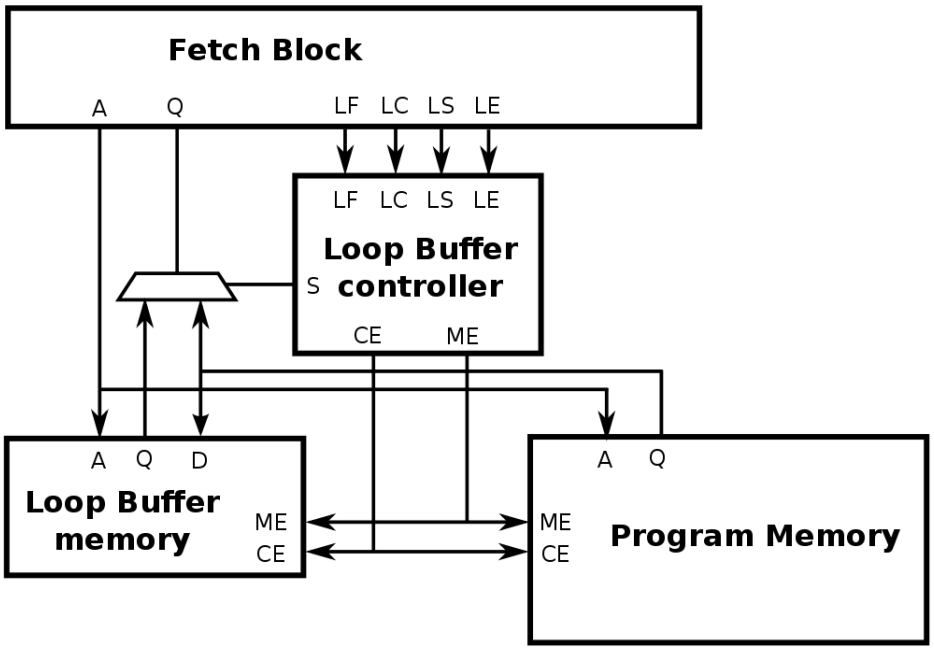


Figure 5.6: IMO interface for a CELB architecture.

In essence, the operation of the loop buffer concept is as follows. During the first iteration of the loop, the instructions are fetched from the program memory to both the loop buffer architecture and the processor architecture. In this iteration, the loop buffer architecture records the instructions of the loop. Once the loop is stored, for the rest of iterations, the instructions are fetched from the loop buffer architecture instead of the program memory. In the last iteration, the connection between the processor architecture and the program memory is restored, such that subsequent instructions are fetched from the program memory. During the execution of non-loop parts of the application code, instructions are fetched directly from the program memory.

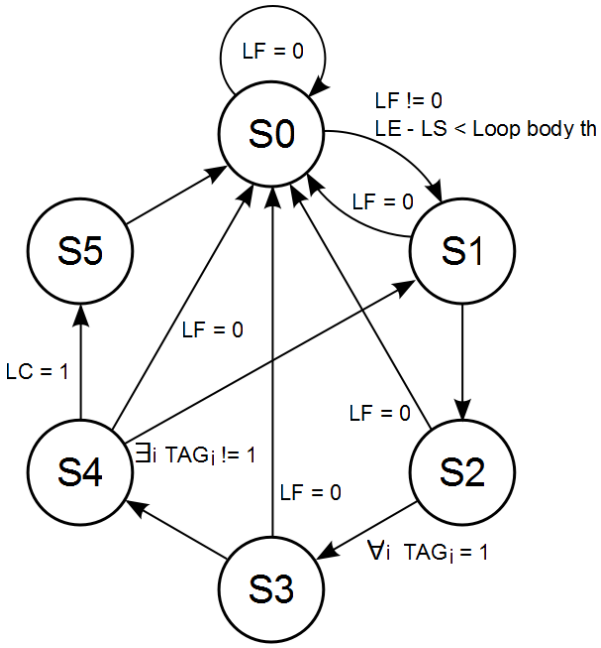


Figure 5.7: State-machine diagram of the loop buffer controller.

The loop buffer controller monitors the operation of the loop buffer architecture based on a state-machine. This state-machine is shown in Figure 5.7. The six states of the state-machine are:

- **s0** Initial state.
- **s1** Transition state between *s0* and *s2*.
- **s2** State where the loop buffer architecture is recording the instructions that the program memory supplies to the processor architecture.
- **s3** Transition state between *s2* and *s4*.
- **s4** State where the loop buffer architecture is providing the instructions to the processor architecture.
- **s5** Transition state between *s4* and *s0*.

The transition states *s1*, *s3*, and *s5* are necessary in order to give the control of the instruction supply from the program memory to the loop buffer architecture and vice-versa. The transition between *s4* and *s1* is necessary because the body size of a loop can change in real-time (*i.e.*, in a loop body with if-statements or function calls). In order to check in real-time whether the loop body size changes or not, a 1-bit tag is used. This tag is associated with each address that is stored in the loop buffer memory. The loop buffer controller checks this tag to know if the address is already stored in the loop buffer architecture or not.

Figure 5.8 shows how the BCLB architecture is composed of different loop buffer memories. In a BCLB architecture, every memory is connected to the processor architecture and the program memory through multiplexers. The loop buffer controller, based on the loop body size of the loop that is on execution, decides which of the available loop buffer memories is connected directly with the program memory and the processor architecture. The logic circuit that decides if the loop buffer architecture is activated is the same as the one that is used in the CELB architecture. In order to make all the decisions that are described previously, the complexity of the state-machine is incremented. However, Figure 5.8 shows that this modification allows the design of the loop buffer architecture to be scalable.

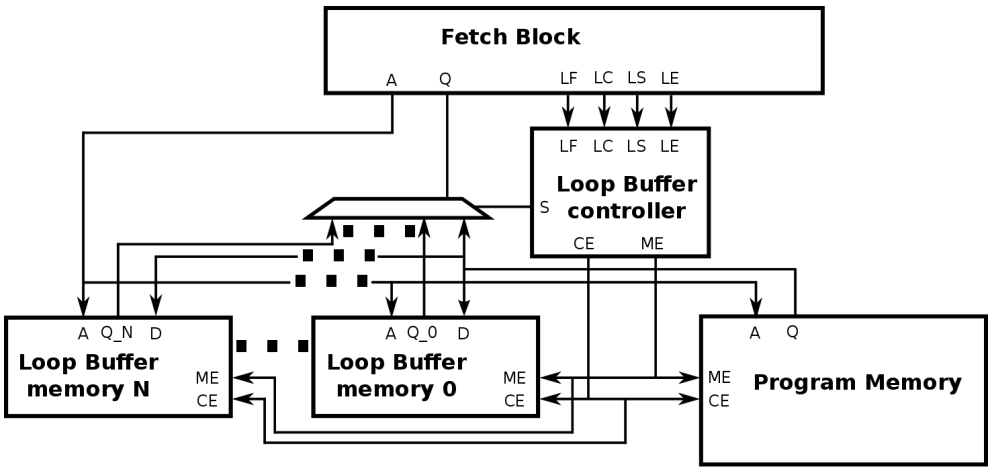


Figure 5.8: IMO interface for a BCLB architecture.

5.4. Experimental Evaluation

This Section shows the results of the experimental evaluation of the CELB architecture and the BCLB architecture. Firstly, Section 5.4.1 describes the methodology that is used in the energy simulations. Secondly, Section 5.4.2 analyses the experimental applications that are described in Section C.3 and Section C.6 based on profiling information. Finally, Section 5.4.3 shows and discusses the results of the energy simulations.

5.4.1. Simulation Methodology

The simulation methodology that is used in the experimental evaluation is described by the following steps:

Application mapping. The selected application is mapped to the system architecture that it is going to be simulated. The I/O data connections of the system are used by the embedded systems designer to corroborate the correct functionality of the system.

Behaviour simulation. The mapped application is simulated on the system architecture in order to check its correct functionality. For that purpose, the ISS (*Instruction Set-Simulator*) from the *Target Compiler Technologies* tools [TAR12] is used.

RTL implementation. The RTL (*Register-Transfer Level*) language description files of the processor architecture are automatically generated using the HDL generation tool from the *Target Compiler Technologies* tools [TAR12]. The design of the interfaces between the DMH and the IMO has to be added in order to have a complete description of the whole system in RTL language.

RTL synthesis. Once every component of the system has its own RTL language description file, the design is synthesised. In our RTL synthesis, a TSMC 90nm LP library is used for a system frequency of 100MHz. During synthesis, clock gating is used whenever possible.

Place and route. After the synthesis, place and route is performed using *Encounter* [CAD12].

Recording Activity. It is necessary to generate a VCD (*Value Change Dump*) file for the desired time interval of the netlist simulation. If the selected time interval is the execution time of the application, the VCD file will contain the information of the activity of every net and every component of the whole system when an input data frame is processed.

Extraction of power consumption. As a final step, the information of the average power consumption is extracted with the help of *Primetime* [SYN12].

Figure 5.9 shows the inputs and outcomes of each step described above.

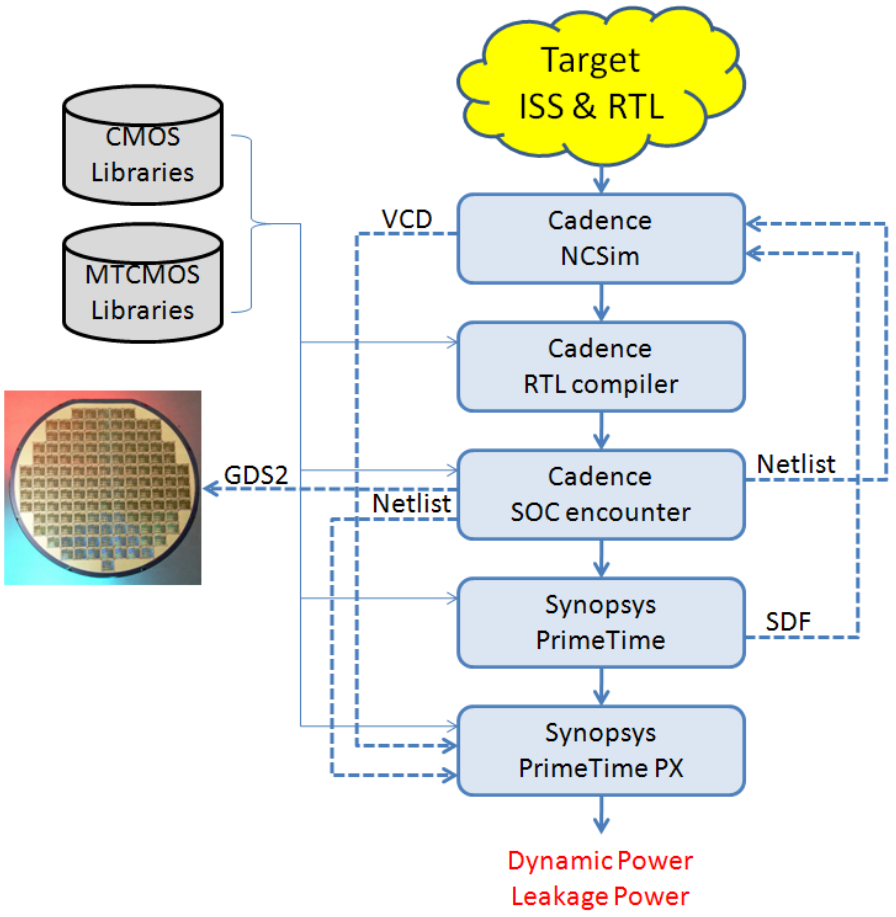


Figure 5.9: Simulation methodology.

5.4.2. Analysis of the Experimental Applications

The total energy consumption of the systems that are presented in this Chapter is strongly influenced by the consumption of the IMO. Following the steps that are described in Section 5.4.1, Figure 5.10, Figure 5.11, Figure 5.12, and Figure 5.13 present the first outcome from the experimental evaluation. Figure 5.10 and Figure 5.11 show the power breakdowns that are related to the heartbeat detection algorithm, whereas Figure 5.12 and Figure 5.13 show the power breakdowns that are related to the AES algorithm. In these Figures, the components of the processor core are grouped. Apart from seeing how the power distribution changes from a design based on a general-purpose processor to an ASIP design, these Figures demonstrate that the total energy consumption of these systems is strongly influenced by the consumption of the IMO.

Loops dominate the total energy consumption of the IMO. Figure 5.14, Figure 5.15, Figure 5.16, and Figure 5.17 show profiling information based on the accesses that are done in the program address space. Figure 5.14 and Figure 5.15 show the profiles based on the number of cycles per program counter that are related to the heartbeat detection algorithm, whereas Figure 5.16 and Figure 5.17 show the profiles based on the number of cycles per program counter that are related to the AES algorithm. It is possible to see from these Figures that there are regions that are more frequently accessed than others. This situation implies the existence of loops. Apart from this fact, it is possible to see from these Figures that the application execution time of the selected applications is dominated by only a few loops.

In order to implement energy efficient IMOs based on loop buffer architectures, more detail information related to loops is needed. Table 5.1, Table 5.2, Table 5.3, and Table 5.4 provide this information. Table 5.1 and Table 5.2 present the loop profiling information of the systems that are related to the heartbeat detection algorithm, whereas Table 5.3 and Table 5.4 present the loop profiling information of the systems that are related to the AES algorithm. In these Tables, loops are numbered in the same order that they appear in the assembly code of the algorithm. A nested loop creates another level of numbering. Thus, a loop named *2* corresponds to the second loop encountered, while a loop named *2.1* corresponds to the first sub-loop encountered in the loop named *2*. These Tables corroborate the fact that the execution time of the loops dominates the total execution time of the application. For instance, the execution time of the loops represents approximately 79 % of the total execution time of the heartbeat detection algorithm in the case of the general-purpose processor, and 81 % in the processor architecture that is optimised for this algorithm. In contrast, in the AES algorithm, the execution time of the loops represents 77 % of the total execution time in the case of the general-purpose processor, and 90 % in the processor architecture that is optimised for this algorithm. It is necessary to remark that differences exist between algorithms of the same application due to the source code transformations and the mapping optimisations that are applied in the optimised algorithms in order to generate efficient codes.

The configurations of the CELB architecture and the BCLB architecture that are analysed in this Chapter are based on the loop profiling presented in Table 5.1, Table 5.2, Table 5.3, and Table 5.4. On the one hand, the selection of the CELB configurations is based on the small size of the loops that have bigger percentage of execution time. With this strategy, it is assumed that these configurations are the most energy efficient. This assumption is based on the fact that these configurations provide the highest energy savings among all the possible configurations. These major energy savings help to reduce the penalty

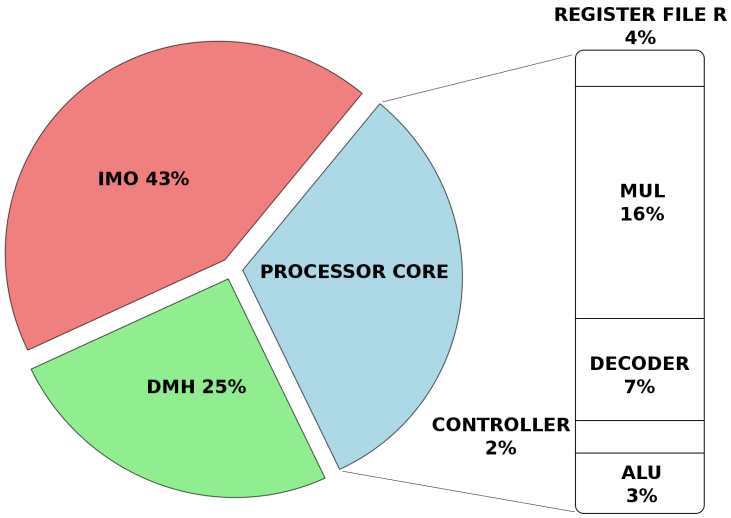


Figure 5.10: Power breakdown in the general-purpose processor running the heartbeat detection algorithm.

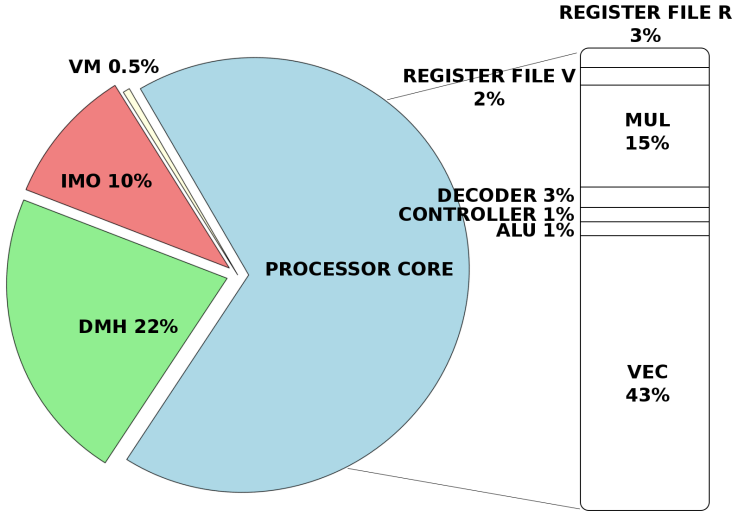


Figure 5.11: Power breakdown in the optimised processor running the heartbeat detection algorithm.

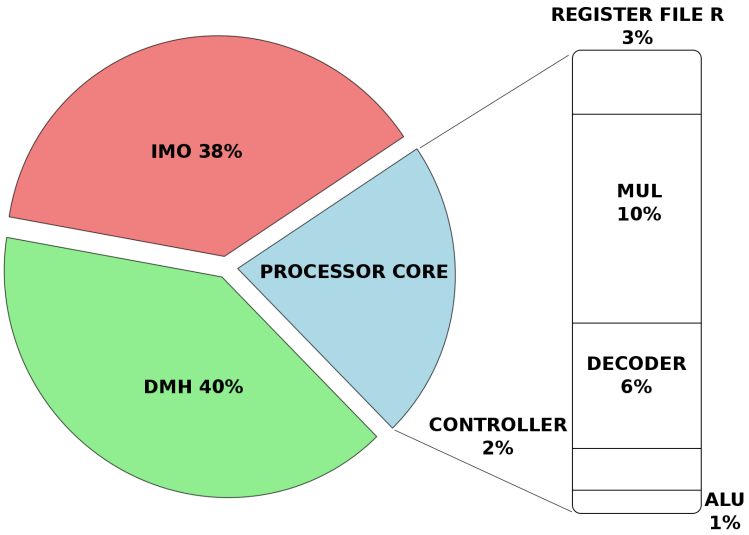


Figure 5.12: Power breakdown in the general-purpose processor running the AES algorithm.

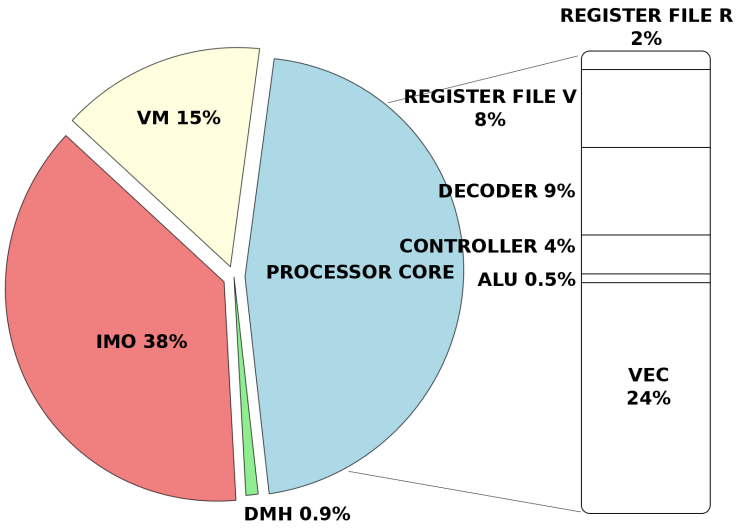


Figure 5.13: Power breakdown in the optimised processor running the AES algorithm.

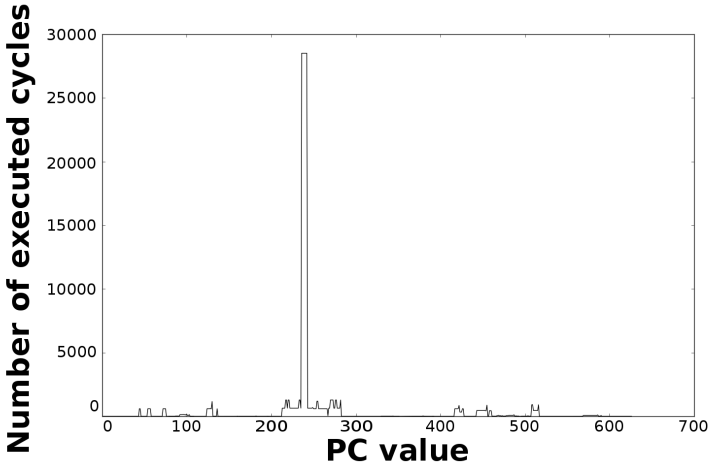


Figure 5.14: Number of cycles per PC in the general-purpose processor running the heartbeat detection algorithm.

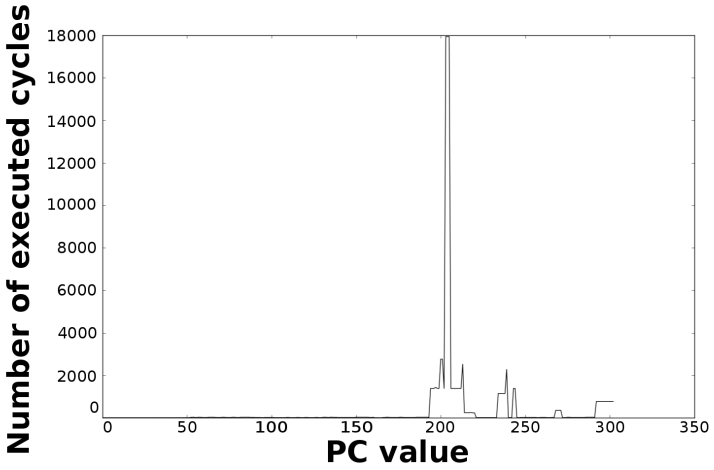


Figure 5.15: Number of cycles per PC in the optimised processor running the heartbeat detection algorithm.

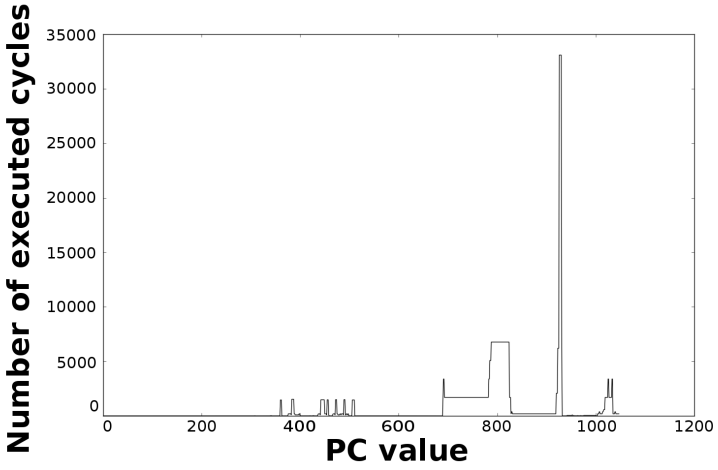


Figure 5.16: Number of cycles per PC in the general-purpose processor running the AES algorithm.

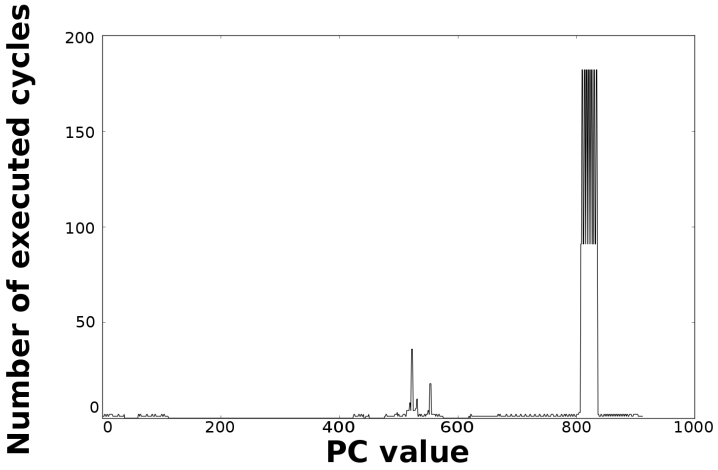


Figure 5.17: Number of cycles per PC in the optimised processor running the AES algorithm.

related to the introduction of the loop buffer architecture in the system. On the other hand, the selection of the BCLB configurations is based on the strategy of taking the maximum loop body size of the application, and chop it by the granularity of the smaller loop body size that the applications contains. This strategy is used in these architectures, because the exact energy consumption of the extra logic that has to be added in the loop buffer controller is unknown. Table 5.5 presents the initial configurations that are evaluated.

In order to conclude the analysis of the experimental applications, it is necessary to remark that due to time requirements, a system frequency of 100MHz is fixed. On the one hand, the heartbeat detection algorithm running on the general-purpose processor spends 462 cycles to process an input sample contained in the data frame. However, if this algorithm is running on the processor optimised for this algorithm, the number of cycles to process the same input sample is 11 cycles. On the other hand, the AES algorithm running on the general-purpose processor spends 484 cycles to process an input sample contained in the data frame. Nevertheless, if this algorithm is running on the processor optimised for this algorithm, the number of cycles to process the same input sample is only 3 cycles.

Table 5.1: Loop profiling of the heartbeat detection algorithm on the general-purpose processor.

	Start address	End address	Loop body size	Number of iterations	Execution time [%]
Loop 1	33	34	2	4	0
Loop 2	44	45	2	594	0
Loop 3	54	57	4	594	1
Loop 4	72	75	4	594	1
Loop 5	92	103	12	132	1
Loop 6	124	136	13	594	3
Loop 7	160	160	1	15	0
Loop 8	236	242	7	32,625	71
Loop 9	417	427	11	594	2
Loop 10	569	590	22	64	0

Table 5.2: Loop profiling of the heartbeat detection algorithm on the optimised processor.

	Start address	End address	Loop body size	Number of iterations	Execution time [%]
Loop 1	192	244	53	1,380	70
Loop 1.1	200	205	6	1	0
Loop 2	266	271	6	350	2
Loop 3	209	302	13	768	9

Table 5.3: Loop profiling of the AES algorithm on the general-purpose processor.

	Start address	End address	Loop body size	Number of iterations	Execution time [%]
Loop 1	307	309	3	8	0
Loop 2	324	327	4	2	0
Loop 3	340	342	3	16	0
Loop 4	360	362	3	1,460	3
Loop 5	383	387	5	1,600	7
Loop 6	409	411	3	4	0
Loop 7	419	421	3	8	0
Loop 8	426	428	3	16	0
Loop 9	436	458	23	92	2
Loop 10	472	474	3	1,392	3
Loop 11	489	491	3	1,392	3
Loop 12	506	510	5	1,460	6
Loop 13	519	523	5	4	0
Loop 14	926	930	5	6,016	25
Loop 15	942	1,000	59	40	2
Loop 16	1,019	1,034	16	1,692	26

Table 5.4: Loop profiling of the AES algorithm on the optimised processor.

	Start address	End address	Loop body size	Number of iterations	Execution time [%]
Loop 1	519	524	6	36	5
Loop 2	544	560	17	2	1
Loop 2.1	550	555	6	0	0
Loop 3	806	837	32	91	84

Table 5.5: Configurations of the experimental framework.

	Baseline architecture	CELB	BCLB
HBD algorithm General-purpose processor	No loop buffer architecture	8 words	8 banks of 8 words
HBD algorithm Optimised processor	No loop buffer architecture	64 words	8 banks of 8 words
AES algorithm General-purpose processor	No loop buffer architecture	8 words	4 banks of 8 words
AES algorithm Optimised processor	No loop buffer architecture	32 words	4 banks of 8 words

5.4.3. Power Analysis

Table 5.6, Table 5.7, and Table 5.8 present the power results for each system that is evaluated. These tables show the dynamic power, the leakage power, and the total power for all the configurations that are presented in Table 5.5. As can be seen, the power consumption of the IMO is the sum of the power that is consumed by the components that the IMO contains (*i.e.*, the loop buffer controller, the loop buffer memory, and the program memory).

Table 5.6: Power consumption [W] of the baseline architecture.

	Component	Dynamic power	Leakage power	Total power
HBD algorithm	IMO	4.44×10^{-06}	0.91×10^{-09}	4.44×10^{-06}
-	LB Controller	0	0	0
General-purpose	LB Memory	0	0	0
processor	PM	4.44×10^{-06}	0.91×10^{-09}	4.44×10^{-06}
HBD algorithm	IMO	3.57×10^{-07}	8.46×10^{-11}	3.57×10^{-07}
-	LB Controller	0	0	0
Optimised	LB Memory	0	0	0
processor	PM	3.57×10^{-07}	8.46×10^{-11}	3.57×10^{-07}
AES algorithm	IMO	1.81×10^{-06}	4.32×10^{-10}	1.82×10^{-06}
-	LB Controller	0	0	0
General-purpose	LB Memory	0	0	0
processor	PM	1.81×10^{-06}	4.32×10^{-10}	1.82×10^{-06}
AES algorithm	IMO	1.20×10^{-06}	2.11×10^{-10}	1.20×10^{-06}
-	LB Controller	0	0	0
Optimised	LB Memory	0	0	0
processor	PM	1.20×10^{-06}	2.11×10^{-10}	1.20×10^{-06}

Table 5.7: Power consumption [W] of the IMO based on an CELB architecture.

	Component	Dynamic power	Leakage power	Total power
HBD algorithm	IMO	1.74×10^{-06}	1.14×10^{-09}	1.74×10^{-06}
-	LB Controller	2.55×10^{-07}	1.60×10^{-10}	2.55×10^{-07}
General-purpose	LB Memory	6.97×10^{-08}	6.60×10^{-11}	6.97×10^{-08}
processor	PM	1.41×10^{-06}	9.16×10^{-10}	1.41×10^{-06}
HBD algorithm	IMO	1.40×10^{-07}	1.77×10^{-10}	1.40×10^{-07}
-	LB Controller	3.71×10^{-08}	2.66×10^{-11}	3.71×10^{-08}
Optimised	LB Memory	5.76×10^{-08}	6.56×10^{-11}	5.76×10^{-08}
processor	PM	4.50×10^{-08}	8.46×10^{-11}	4.51×10^{-08}
AES algorithm	IMO	1.76×10^{-06}	5.25×10^{-10}	1.76×10^{-06}
-	LB Controller	1.03×10^{-07}	7.39×10^{-11}	1.03×10^{-07}
General-purpose	LB Memory	9.54×10^{-09}	2.68×10^{-11}	9.54×10^{-09}
processor	PM	1.65×10^{-06}	4.25×10^{-10}	1.65×10^{-06}
AES algorithm	IMO	8.32×10^{-07}	4.12×10^{-10}	8.36×10^{-07}
-	LB Controller	2.43×10^{-07}	7.53×10^{-11}	2.47×10^{-07}
Optimised	LB Memory	1.79×10^{-07}	1.29×10^{-10}	1.79×10^{-07}
processor	PM	4.10×10^{-07}	2.13×10^{-10}	4.10×10^{-07}

Table 5.8: Power consumption [W] of the IMO based on a BCLB architecture.

	Component	Dynamic power	Leakage power	Total power
HBD algorithm	IMO	1.97×10^{-06}	1.47×10^{-09}	1.97×10^{-06}
-	LB Controller	4.72×10^{-07}	3.95×10^{-10}	4.72×10^{-07}
General-purpose	LB Memory	8.73×10^{-08}	1.59×10^{-10}	8.73×10^{-08}
processor	PM	1.41×10^{-06}	9.16×10^{-10}	1.41×10^{-06}
HBD algorithm	IMO	1.64×10^{-07}	3.83×10^{-10}	1.65×10^{-07}
-	LB Controller	5.51×10^{-08}	1.40×10^{-10}	5.51×10^{-08}
Optimised	LB Memory	6.39×10^{-08}	1.58×10^{-10}	6.39×10^{-08}
processor	PM	4.50×10^{-08}	8.46×10^{-11}	4.51×10^{-08}
AES algorithm	IMO	1.90×10^{-06}	7.40×10^{-10}	1.90×10^{-06}
-	LB Controller	2.35×10^{-07}	2.72×10^{-10}	2.35×10^{-07}
General-purpose	LB Memory	1.46×10^{-08}	4.29×10^{-11}	1.46×10^{-08}
processor	PM	1.65×10^{-06}	4.25×10^{-10}	1.65×10^{-06}
AES algorithm	IMO	6.60×10^{-07}	4.30×10^{-10}	6.60×10^{-07}
-	LB Controller	5.20×10^{-08}	1.10×10^{-11}	5.20×10^{-08}
Optimised	LB Memory	1.98×10^{-07}	2.06×10^{-10}	1.98×10^{-07}
processor	PM	4.10×10^{-07}	2.13×10^{-10}	4.10×10^{-07}

It is possible to see from these Tables that the systems that are optimised for the experimental applications always consume less power than the general-purpose systems. Therefore, the introduction of the CELB architecture and the BCLB architecture does not affect this energy consumption trend.

Analysing Table 5.7, it is possible to see that there is a decrease on the dynamic power of these systems in relation to the baseline architectures. This is because the majority of the instructions are fetched from a small memory instead of the large memory that forms the program memory. On the other hand, the CELB architectures have an increase in the leakage power consumption in relation to the baseline architectures, due to the introduction of the loop buffer architecture. It is also possible to see the importance of the loop buffer controller in the IMO, which accounts from the 5 % of the power consumption of the IMO in the system where the AES algorithm is running on the general-purpose processor, to 30 % in the system where the AES algorithm is running on the processor architecture optimised for this algorithm.

Using the profiling information presented in Table 5.1, Table 5.2, Table 5.3, and Table 5.4, and the power results obtained from the simulations of the systems presented in Table 5.5, it is possible to evaluate whether the initial configurations for the CELB architecture are selected correctly from the energy consumption point of view.

For the heartbeat detection algorithm running on the general-purpose processor, Figure 5.18 shows the power reductions that can be achieved for all the possible configurations. In the configuration of 8 words, the 73 % of the execution time of the application is on loops, while in the rest of

the configurations this percentage is 79 %. It is possible to see that in this scenario, the best configuration is a loop buffer memory of 16 words, because the increase of the use of the loop buffer memory compensates the penalty introduced by using a bigger loop buffer architecture.

Figure 5.19 shows the power reductions that can be achieved for all the possible configurations when the heartbeat detection algorithm is running on the processor architecture optimised for this algorithm. In the configuration of 8 words, the 2 % of the execution time of the application is on loops; this percentage is 11 % in the configuration of 16 and 32 words; whereas in the configuration of 64 words this percentage is 81 %. It is possible to see that in this scenario, the only configuration that brings energy savings is the loop buffer memory of 64 words. The percentages of the execution time of the rest of configurations do not compensate the penalty introduced by using a loop buffer architecture.

For the AES algorithm running on the general-purpose processor, Figure 5.20 shows the power reductions that can be achieved for all the possible configurations. In the configuration of 8 words, the 47 % of the execution time of this application is on loops; in the configuration of 16 words this percentage is 70 %; in the configuration of 32 words this percentage is 75 %; whereas in the configuration of 64 words this percentage is 77 %. It is possible to see that in this scenario, the best configuration is a loop buffer memory of 32 words, because the increase of the use of the loop buffer architecture compensates the penalty introduced by using a bigger loop buffer memory. Besides, the small increase in the percentage of execution time from the configuration of 32 words to 64 words does not compensate the increase in the leakage power consumption that this last loop buffer architecture has.

Figure 5.21 shows the power reductions that can be achieved for all the possible configurations when the AES algorithm is running on the processor architecture optimised for this algorithm. In the configuration of 8 words, the 5 % of the execution time of the application is on loops; in the configuration of 16 words this percentage is 6 %; whereas in the configuration of 32 and 64 words this percentage is 90 %. It is possible to see that in this scenario, the best configuration is a loop buffer memory of 32 words. The percentages of the execution time of the application for the 8 and the 16 words configurations do not compensate the penalty introduced by using a loop buffer architecture. Also in this scenario, the small increase in the percentage of the execution time of the application from the configuration of 32 words to 64 words does not compensate the increase in the leakage power consumption that this last loop buffer architecture has.

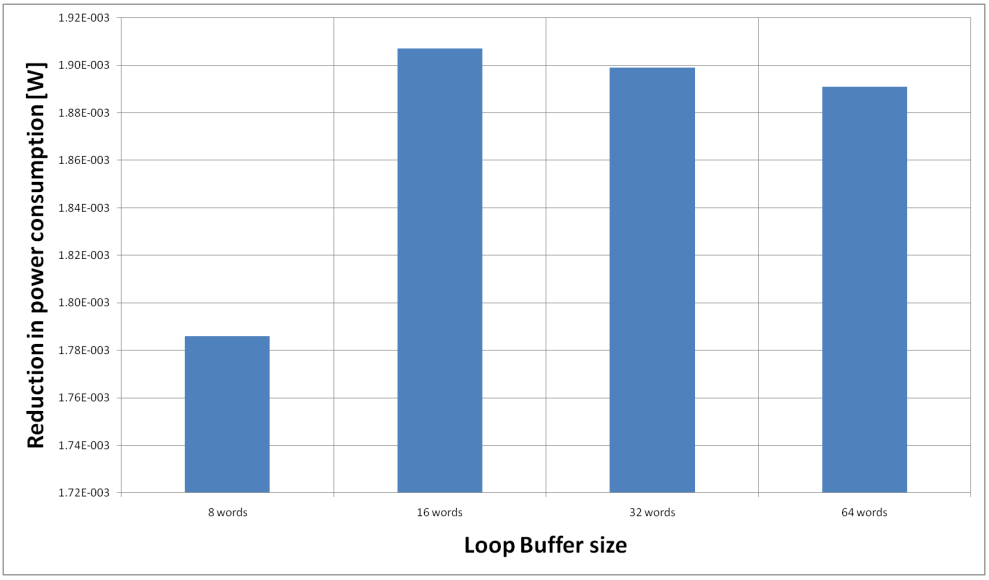


Figure 5.18: HBD algorithm running on the general-purpose processor using different configurations for the CELB architecture.

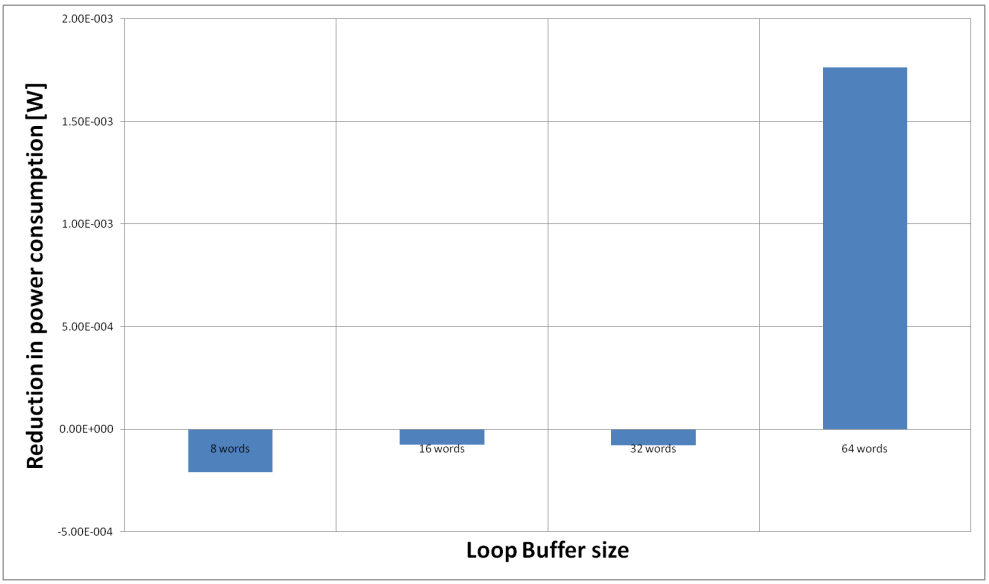


Figure 5.19: HBD algorithm running on the optimised processor using different configurations for the CELB architecture.

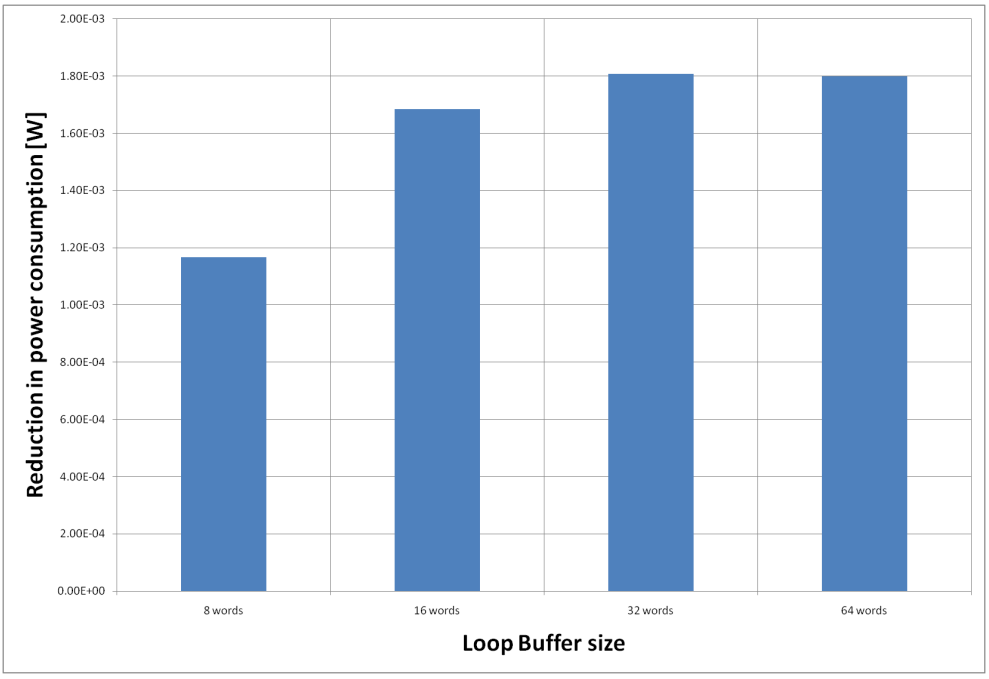


Figure 5.20: AES algorithm running on the general-purpose processor using different configurations for the CELB architecture.

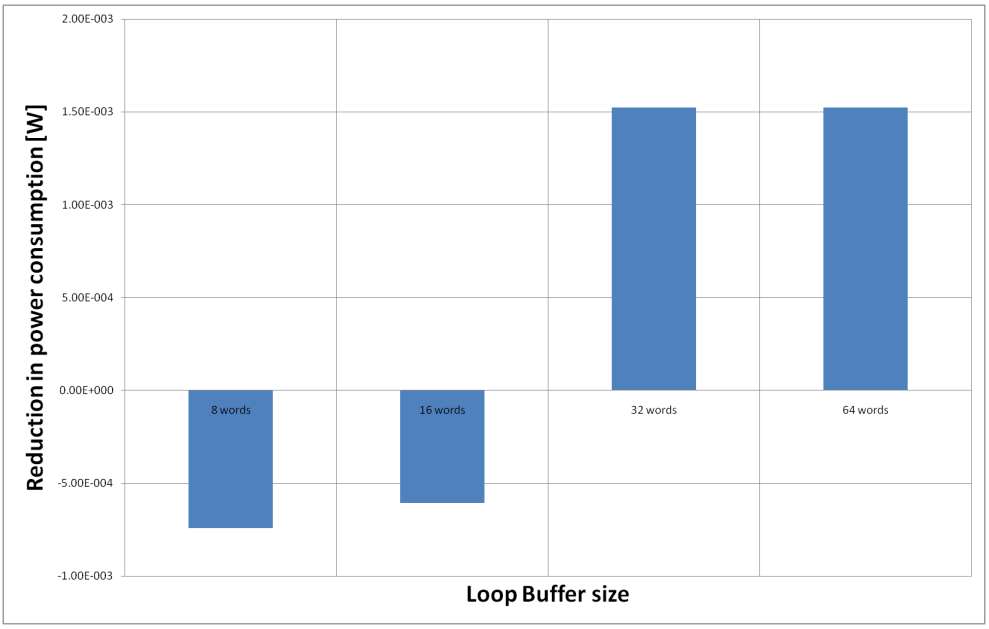


Figure 5.21: AES algorithm running on the optimised processor using different configurations for the CELB architecture.

Analysing Table 5.8, it is possible to see that also in these architectures, there is a decrease in the dynamic power consumption of these systems in relation to the baseline architectures. However, it is possible to see that these architectures sometimes do not offer as good energy savings as the CELB architectures offer, because the system suffers an increase in both the dynamic and the leakage power consumption with the introduction of these loop buffer architectures. Firstly, in the dynamic power consumption, the loop buffer controller of the BCLB architecture has higher complexity than in the CELB architecture. Secondly, in the leakage power consumption, apart from the higher complexity of the loop buffer controller, there is more loop buffer memories. In these loop buffer architectures, the importance of the loop buffer controller is increased in the IMO, which now accounts for 10 % of the power consumption of the IMO in the AES algorithm when it is running on the general-purpose processor, and for 32 % in the heartbeat detection algorithm running on the processor optimised for this algorithm. Using the same information and methodology as in the analysis of the CELB architectures, it is possible to analyse whether the selected configurations for the BCLB architectures are energy efficient.

For the heartbeat detection algorithm running on the general-purpose processor architecture, it is necessary to analyse only the loop buffer configurations that have 8 words, because all the loops can fit in a loop buffer memory of 16 words (see Table 5.1), and every configuration in a BCLB architecture with a loop buffer memory of 16 words is worse in terms of power consumption than a CELB architecture with a loop buffer memory of 16 words. Figure 5.22 shows the possible configurations of two loop buffer memories, where one of them has a fixed size of 8 words. From this Figure, it is possible to see that the best configuration is two loop buffer memories of 8 words each. If the energy savings from the BCLB architecture and the CELB architecture are compared, it is possible to see that for this specific scenario, it is better to have the CELB architecture.

For the heartbeat detection algorithm running on the processor architecture optimised for this algorithm, it is necessary to analyse only the loop buffer configurations that have 64 words, because any configuration without a loop buffer memory of this size will not bring energy savings (see Figure 5.19). Figure 5.23 shows the configuration of two loop buffer memories, where one of them has a fixed size of 64 words. From this Figure, it is possible to see that the best configuration is a loop buffer memory of 16 words together with the loop buffer memory of 64 words. If the energy savings from the BCLB architecture and the CELB architecture are compared, it is possible to see that for this specific scenario it is also better to have the CELB architecture.

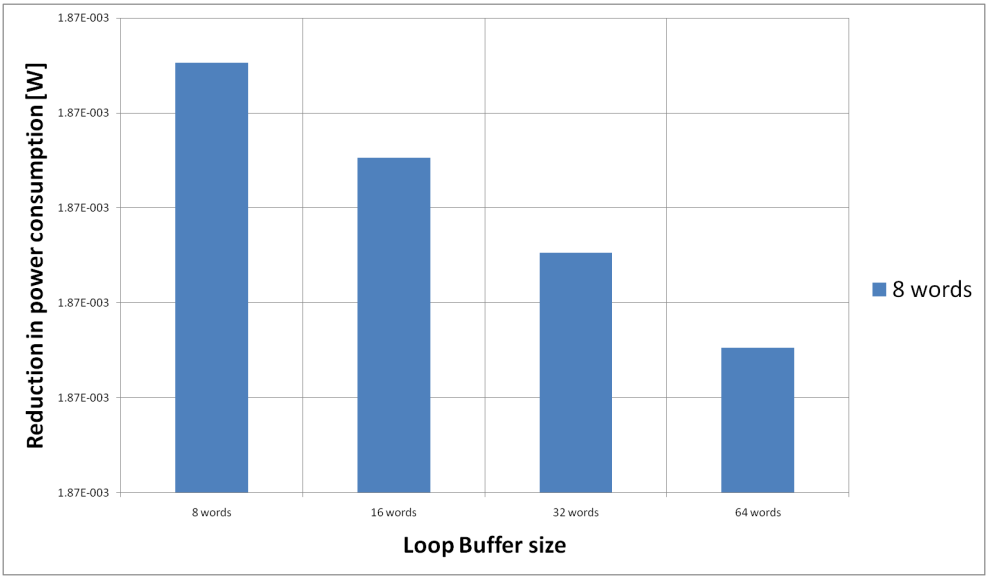


Figure 5.22: HBD algorithm running on the general-purpose processor using different configurations for the BCLB architecture.

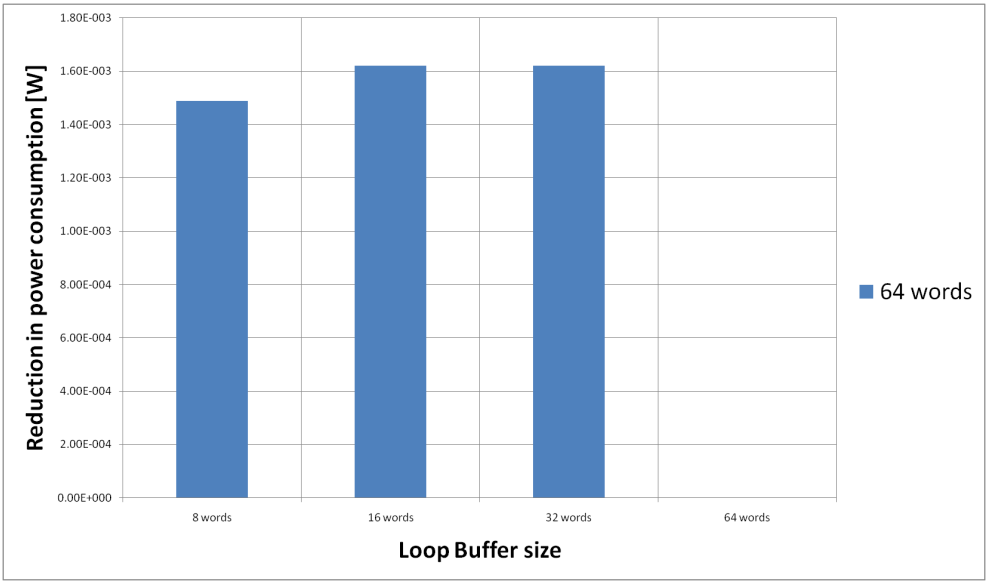


Figure 5.23: HBD algorithm running on the optimised processor using different configurations for the BCLB architecture.

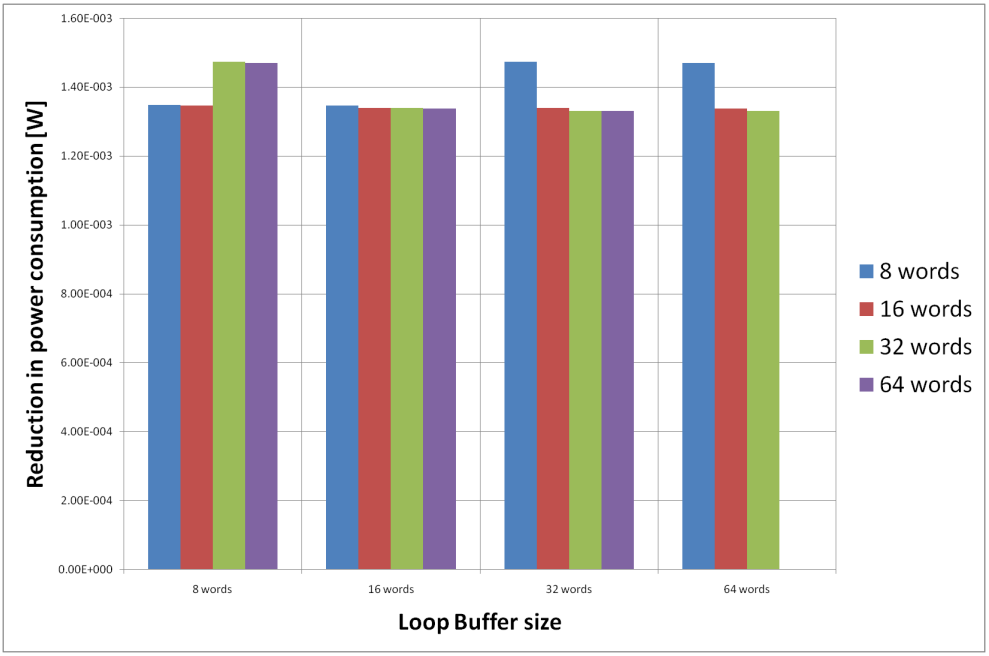


Figure 5.24: AES algorithm running on the general-purpose processor using different configurations for the BCLB architecture.

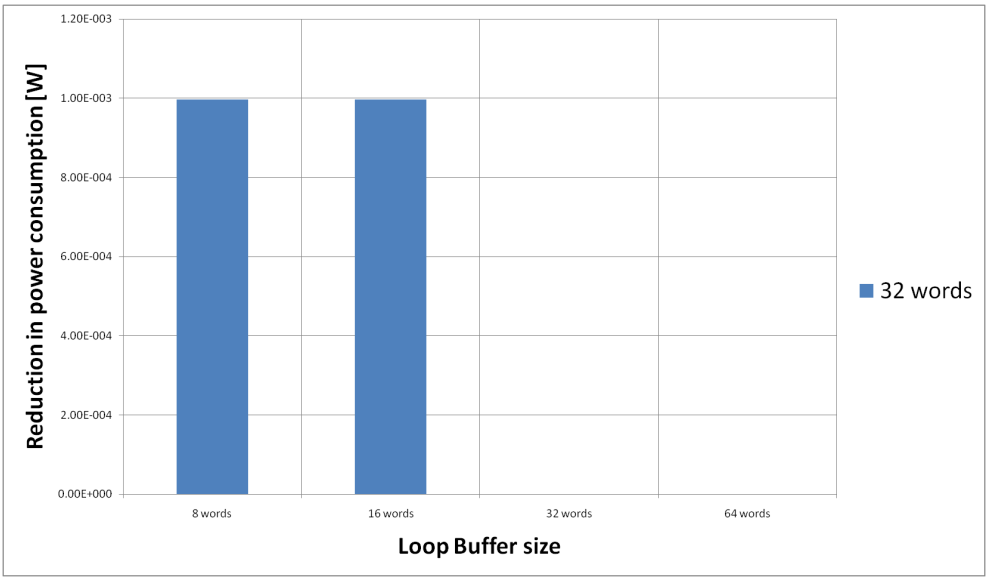


Figure 5.25: AES algorithm running on the optimised processor using different configurations for the BCLB architecture.

For the AES algorithm running on the general-purpose processor architecture, it is necessary to analyse all the possible configurations, because the execution time of the application is spread (see Table 5.3). The configuration with two loop buffer memories of 64 words each is not analysed, because this configuration is worse in energy efficiency than the CELB architecture of 64 words, due to the increase in energy consumption of the loop buffer controller. From Figure 5.24, it is possible to see that the best configuration is a loop buffer of 8 words together with a loop buffer of 32 words. In this case, if the energy savings from the BCLB architecture and the CELB architecture are compared, it is possible to see that for this specific scenario it is also better to have the CELB architecture.

For the AES algorithm running on the processor architecture that is optimised for this algorithm, it is necessary to analyse only the loop buffer configurations that have 32 words, because all the loops can fit in a loop buffer memory of 32 words (see Table 5.4). However, from Figure 5.21, it is possible to see that only loop buffer memories of 32 and 64 words bring energy savings. Therefore, the analyse will be focused only on the loop buffer configurations that has 32 words. Figure 5.25 shows the configuration of two loop buffer memories, where one of them has a fixed size of 32 words. From this Figure, it is possible to see that the best configuration is a loop buffer memory of 8 words together with the loop buffer memory of 32 words. If the energy savings from the BCLB architecture and the CELB architecture are compared, it is possible to see that for this specific scenario it is also better to have the CELB architecture.

Based on all the previous results and discussions, it is possible to conclude that the use of loop buffer architectures in order to optimise the IMO from the energy efficiency point of view should be evaluated carefully. In the case studies that are presented in this Chapter, the CELB architecture is normally more energy efficient than the BCLB architecture, as can be seen in Figure 5.26. However, the CELB architecture is not always more energy efficient than the BCLB architecture. The higher energy efficiency of the CELB architecture is due to the fact that the whole execution time of all benchmarks is concentrated in a few loops with similar loop body size. If a benchmark can be found, in which this percentage is shared between loops with different loop body sizes, the BCLB architecture will then bring more energy efficiency than the CELB architecture. Therefore, the two factors that have to be taken in account in order to implement an energy efficient IMO based on a loop buffer architecture are:

- the percentage of the execution time of the application that is related to the execution of the loops that are included in the application. If this percentage is low, the introduction of a loop buffer architecture in the IMO will not offer any energy savings, because the loop buffer

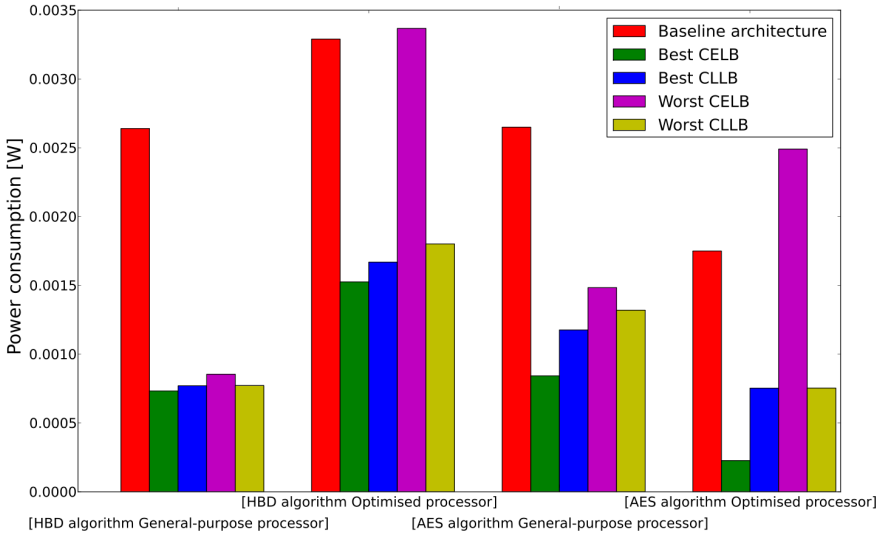


Figure 5.26: Summary of the best and worst CELB and BCLB architectures.

architecture is not used enough to achieve energy savings. In contrast, the higher this percentage, the higher energy savings that can be achieved.

- the distribution of the execution time percentage, which is related to the execution of the loops, over each of the loops that form the application. For instance, the whole execution time percentage that is related to loops can belong only to a few loops, or in another case, this percentage can be spread in each loop homogeneously. If the whole execution time is concentrated in a few loops, the CELB architecture will bring more energy savings than the BCLB architecture. If this percentage is distributed homogeneously among loops, the BCLB architecture will then bring more energy savings than the CELB architecture. These facts are based on the efficient use of the multi-banks that can form the loop buffer architecture.

5.5. Conclusions

In this Chapter, the loop buffer concept was applied in two real-life embedded applications that are widely used in the nodes of biomedical WSNs. The loop buffer architectural organisations that were analysed in this Chapter were the CELB (*Central Loop Buffer Architecture for Single Processor Organisation*) and the BCLB (*Banked Central Loop Buffer Architecture*). An analysis of the experimental applications that were used in this Chapter

was performed to show which type of loop buffer scheme was more suitable for applications with certain behaviour. To evaluate the power impact, post-layout simulations were used to have an accurate estimation of parasitics and switching activity. The evaluation was performed using a TSMC 90nm LP library and commercial memories. From the experimental evaluation, gate-level simulations demonstrated that a trade-off exists between the complexity of the loop buffer architecture and the power benefits of utilising it. This confirms the results, showing that the BCLB architecture did not always bring benefits. Therefore, the use of loop buffer architectures in order to optimise the IMO from the energy efficiency point of view should be evaluated carefully. Two factors have to be taken into account in order to implement an energy efficient IMO based on a loop buffer architecture: (1) the percentage of the execution time of the application that is related to the execution of the loops that are included in the application, and (2) the distribution of the execution time percentage, which is related to the execution of the loops, over each of the loops that form the application.

In the next chapter...

the reader will find the analysis of non-overlapping and complementary implementation options for distinct partitions of the design space that is related to distributed loop buffer architectures. The high-level trade-off analysis of the proposed implementations is crucial in order to present the correct process design that an embedded systems designer has to follow in order to have an efficient distributed loop buffer architecture for a certain application.

Chapter 6

Design Space Exploration of Distributed Loop Buffer Architectures with Incompatible Loop-Nest Organisations

"If I had asked people what they wanted, they would have said faster horses."

— Henry Ford.

This Chapter proposes and analyses non-overlapping and complementary implementation options for the distinct partitions of the design space that are related to distributed loop buffer architectures. The high-level trade-off analysis of the proposed implementations is crucial in order to present the correct process design that an embedded systems designer has to follow in order to have an efficient distributed loop buffer architecture for a certain application. Results show that, with an increase of about 6.5 % in the energy consumption of the control logic that exists in the instruction memory organisation, the overall energy consumption of the instruction memory organisation can be reduced from 6 % to 22 %, when distributed loop buffer architectures with incompatible loop-nest organisations are used instead of clustered loop buffer architectures with shared loop-nest organisations architectures.

6.1. Introduction

The design of actual embedded systems is constrained by the requirements of modern embedded applications. These applications require not only sustained operation for long periods of time, but also to be executed on not mains-connected systems. Under the constraint of not being mains-connected, the

absence of wires to supply a constant source of energy causes that the use of an energy harvesting source [CC08] or an integrated energy supplier (*e.g.*, a battery) limits the operation time of these devices. In order to achieve the low-power constraints that modern embedded applications require, it is crucial not only to decrease the total energy consumption of all parts of the system, but also to have a better distribution of the energy budget throughout the whole system. Therefore, embedded systems designers have to look at the complete system and tackle the energy consumption problem in each part of the system.

Previous works like [CRL+10] and [VM07] have demonstrated that, in embedded systems, the IMO (*Instruction Memory Organisation*) takes significant portions of chip area and energy consumption. The importance of the energy bottleneck of the IMO becomes more apparent after techniques like loop transformations, software controlled caches, and instruction layout optimisations have been applied [BSL+02, KKC+04]. The state of the art of the architectural enhancements that are used to reduce the energy consumption of the IMO includes the modification or/and the partition of the IMO. On the one hand, loop buffering is a good example of effective scheme for the modification of the hierarchy that exists in the IMO. J. Kin *et al.* [KGMS97] showed that storing small program segments in smaller memory (*e.g.*, in the form of a LB (*Loop Buffer*)), the dynamic energy consumption of the embedded system was reduced significantly. On the other hand, memory banking is a good example of effective method for partitioning the IMO [KKK02]. Apart from the possibility of using multiple low-power operating modes, the use of memory banks reduces the effective capacitance as compared to a single monolithic memory, which leads to further energy reductions. From the point of view of an embedded systems designer, the IMO is specially an issue when large and wide application codes are executed in VLIW (*Very Long Instruction Word*) architectures. In these architectures, the IMO is typically centralised and has a low-energy efficiency [JBA+05]. In order to improve both performance and energy efficiency in embedded systems, the effective use of parallelism has to be boosted [Man05]. Hence, there is a need to solve the problem of the energy consumption of the IMO with a distributed and scalable solution that uses more than one thread of control with minimal hardware overhead.

The contribution of this Chapter is to propose and analyse three options to implement the efficient DLB (*Distributed Loop Buffer Architecture with Incompatible Loop-Nest Organisation*) for a given application. It is important to clarify that, for the whole IMO, the embedded systems designer has the option not only to choose one of the proposed implementations for the control logic of the IMO, but also to combine the three proposed implementations in order to achieve the optimal specific configuration that creates the most efficient implementation of the IMO for a given application.

The proposed implementations take into account not only the possible energy savings that can be achieved in the embedded system, but also the required performance of the embedded application as well as the memory area occupancy. Embedded systems designers take decisions in early stages of the design that can dramatically affect the energy consumption of the embedded system. Therefore, the high-level trade-off analysis of the proposed implementations is crucial in order to present the correct process design that an embedded systems designer has to follow in order to have an efficient implementation of the DLB architecture for a certain application. Results from this analysis show that the selection of the enhancement that has to be introduced in the IMO has to be based on the correct decisions in the trade-offs that exist between energy budget, required performance, and area cost of the embedded system.

The rest of the Chapter is organised as follows. Section 6.2 describes the state of the art of the loop buffer concept. Section 6.3 presents not only the DLB architecture, but also an illustration of why this architecture is attractive for the incompatible loop-nest realisation. Section 6.4 presents the details of the architectural implementations that are proposed in this Chapter. Section 6.5 describes the experimental framework, while the high-level trade-off analysis of the proposed implementations is presented in Section 6.6. Finally, the conclusions of this Chapter are presented in Section 6.7.

6.2. Related Work

The CELB (*Central Loop Buffer Architecture for Single Processor Organisation*) represents the most traditional use of the loop buffer concept. For more details see Section 2.3.2. Figure 6.1 shows the generic architecture of this IMO. This architecture neither has partitioning in the loop buffer memory nor in the PM (*Program Memory*), and its connections depend on a single centralised component. Therefore, efficient parallelism in the execution of an application cannot be achieved in this kind of loop buffer architectures due to the lack of hardware resources.

Since loops form the most important part of an application code [VLCV01], techniques like loop fusion and other loop transformations are applied to exploit the parallelism in order to boost ILP (*Instruction-Level Parallelism*) within loops on single threaded architectures. However, the centralised resources and the global communication of single-threaded architectures make the CELB architecture less energy efficient, when loop transformation techniques are applied to these architectures in order to exploit parallelism within loops. The CLLB (*Clustered Loop Buffer Architecture with Shared*

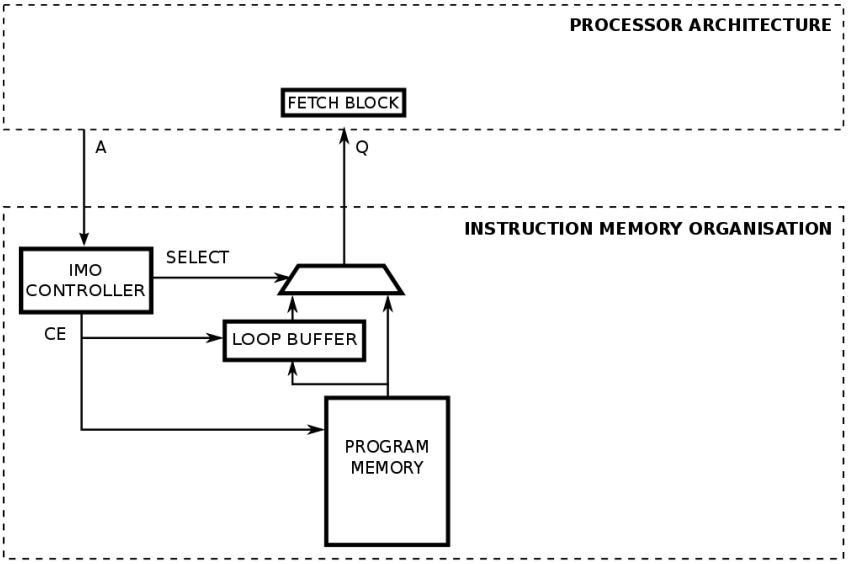


Figure 6.1: Instruction memory organisation with a central loop buffer architecture for single processor organisation.

Loop-Nest Organisation) mitigates these bottlenecks. For more details see Section 2.3.3. Figure 6.2 shows the generic architecture of a CLLB architecture, which inner connections are controlled by one single component. In this architecture, the controller is more complex than the previous one, because it controls the partitions that exist in the loop buffer memory and in the program memory. This set of architectures also includes all the enhancements that come from the introduction of low-power operating modes, as well as from the research that was performed in power management of banked memories [BMP00, FEL01, LK04].

Finally, efficient parallelism exploitation is not yet fully achievable with the CLLB architecture, due to the fact that loops with different threads of control have to be merged into a single loop with a single thread of control. This code transformation is performed using techniques like loop transformations (*e.g.*, loop fusion). However, not all loops of an application can be efficiently exploited by this manner. In the case of incompatible loops, the parallelism cannot be efficiently exploited because they require multiple loop controllers, which results in loss of energy and performance. Therefore a need exists for a multi-threaded platform, that could support execution of multiple incompatible loops, with minimal hardware overhead. That is achievable with the DLB (*Distributed Loop Buffer Architecture with Incompatible Loop-Nest Organisation*). The description of these architectures is shown in the next Section.

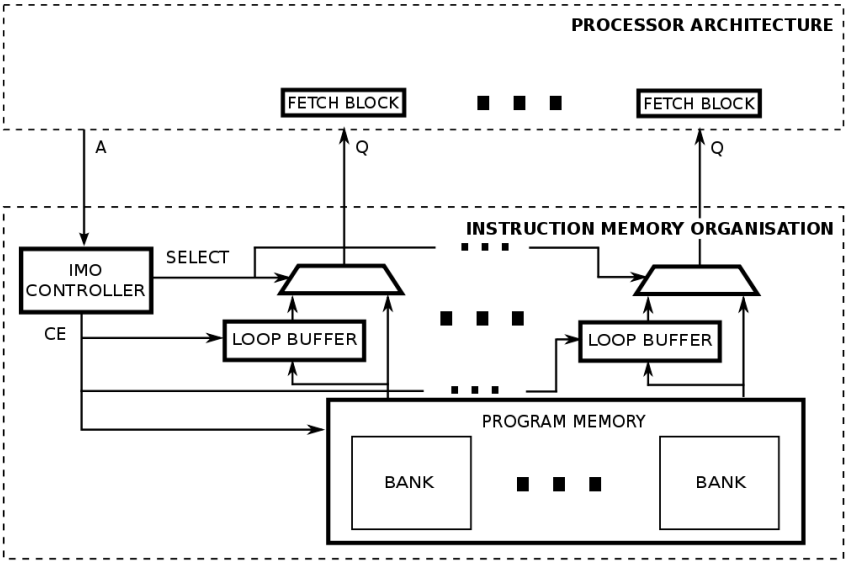


Figure 6.2: Instruction memory organisation with a clustered loop buffer architecture with shared loop-nest organisation.

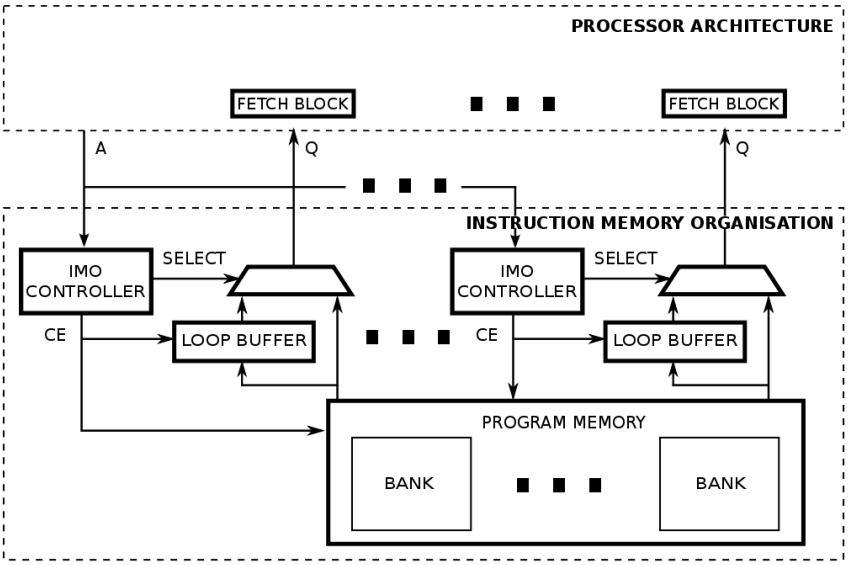


Figure 6.3: Instruction memory organisation with a distributed loop buffer architecture with incompatible loop-nest organisation.

6.3. Motivating Example for Usages of DLB Architectures

This Section presents not only the DLB architecture in Section 6.3.1, but also an illustration of why this loop buffer architecture is attractive for the incompatible loop-nest realisation in Section 6.3.2.

6.3.1. DLB Architecture

In the considered loop buffer architecture not only the loop buffer memories are distributed, but also the loop controllers that manage them. Therefore, in this special set of loop buffer architectures, each loop buffer memory has its own local loop controller. Due to this fact, DLB architectures can work like multi-threaded platforms allowing the execution of incompatible loops in parallel with minimal hardware overhead. For more details see Section 2.3.4. Figure 6.3 shows the generic architecture of this recent representative loop buffer architecture. As shown in this Figure, the inner connections of this IMO are managed by a logic of controllers that is distributed across the architecture. In this IMO, partition exists in both the loop buffer memory and the program memory. The controller of this IMO is even more complex than the controllers of the previous loop buffer architectures, because it also controls the execution of each loop in the corresponding loop buffer memory in order to allow the parallel execution of loops with different iterators.

Table 6.1: Power consumption of loops that are executed over loop buffer memories with different sizes.

Loop Body Size [Instruction Word]	**Loop Buffer Size [Instruction Word]**		
	4	16	32
4	1.02×10^{-04} [W]	1.72×10^{-04} [W]	3.73×10^{-04} [W]
16	1.05×10^{-03} [W]	1.72×10^{-04} [W]	3.73×10^{-04} [W]
32	1.05×10^{-03} [W]	1.10×10^{-03} [W]	3.73×10^{-04} [W]

6.3.2. Motivating Example

Figure 6.4 presents the execution of a realistic illustrative example in the loop buffer architectures that are described in Section 6.2 and Section 6.3.1. This Figure is used to show the benefits and the drawbacks of each one of the loop buffer architectures. Assuming that this benchmark is composed of three loops of 4, 16, and 32 instructions words respectively, its execution in a CELB architecture (see Figure 6.1) can be represented as shown in the case (a) of Figure 6.4. In this Figure, the loops are sequentially mapped during the execution of the benchmark. The loop buffer size is fixed to the size of

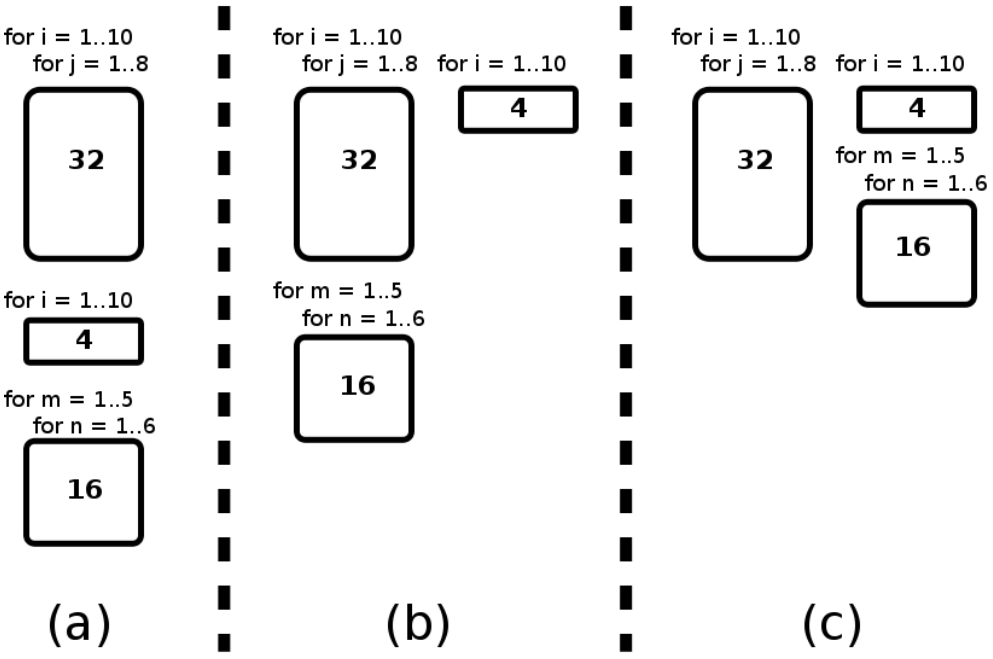


Figure 6.4: Executing a realistic illustrative example in the representative loop buffer architectures.

the bigger loop from the set of loops that compose this benchmark. With this strategy, all loops that form the benchmark are stored without any split. If any split is present in the loops, part of the instructions are fetched from the program memory leading to reduce the energy savings of the loop buffer architecture. Using the values of power consumption that are presented in Table 6.1, the energy consumption of this loop buffer architecture is estimated for a specific system frequency of operation (*i.e.*, 100MHz) in Equation 6.1. Note that in all the calculations, E_{lbXLBY} is the energy that a loop of X instruction words of loop body consumes in a loop buffer architecture with a size of Y instruction words. It is possible to see that the calculation of this energy consumption E_{lbXLBY} is based on three components: the power consumption of the loop buffer architecture, the total number of accesses that are performed over the loop buffer architecture, and the time that requires one access in order to be performed. These three components are split by parenthesis in Equation 6.1, Equation 6.2, Equation 6.3, Equation 6.4, and Equation 6.5 in order to appreciate how they change between different cases.

$$\begin{aligned} E_{CELB} = \; & E_{lb4LB32} + E_{lb16LB32} + E_{lb32LB32} \\ = \; & ((3.73 \times 10^{-04}) \times (4 \times 10) \times (10^{-08})) \\ + \; & ((3.73 \times 10^{-04}) \times (16 \times 30) \times (10^{-08})) \\ + \; & ((3.73 \times 10^{-04}) \times (32 \times 80) \times (10^{-08})) = 1.15 \times 10^{-08} J \end{aligned} \tag{6.1}$$

If this benchmark is executed in a CLLB architecture (see Figure 6.2), case (b) of Figure 6.4, the first step is to analyse the data dependencies between loops and see which loops are incompatible. In this motivating example, because it is assumed that there are no data dependencies, only the loops that have a size of 4 and 32 instruction words can be executed in parallel. In this last case, the program memory and the loop buffer are split in smaller and individual sizes to fit the necessity of each instruction cluster that form the loop buffer architecture. The loop that has a size of 16 instruction words is incompatible with the other two loops, and this loop buffer architecture does not support execution of multiple incompatible loops in parallel. On the one hand, if two loop buffers with the same size are used (*i.e.*, 32 instruction words), the energy consumption is estimated by Equation 6.2. On the other hand, if the choice is to adapt the size of the loop buffers to the loops that are executed in them (*i.e.*, loop buffers of 4 and 32 instruction words), the energy consumption is estimated by Equation 6.3. Based on these results, it is possible to see that the improvement in energy savings, that comes from the use of a CLLB architecture instead of a CELB architecture, is related to the better adaptation of the sizes of the loop buffers to the sizes of the loops that form the application. Besides, in this case, NOP instructions are read from the banks that form the program memory, but not written to the loop buffers, decreasing the transactions between components of the IMO. This introduces a penalty on the performance of our benchmark on the CLLB architecture.

$$\begin{aligned} E_{CLLB1} &= E_{lb4LB32} + E_{lb16LB32} + E_{lb32LB32} \\ &= ((3.73 \times 10^{-04}) \times (4 \times 10) \times (10^{-08})) \\ &+ ((3.73 \times 10^{-04}) \times (16 \times 30) \times (10^{-08})) \\ &+ ((3.73 \times 10^{-04}) \times (32 \times 80) \times (10^{-08})) = 1.15 \times 10^{-08} J \end{aligned} \tag{6.2}$$

$$\begin{aligned} E_{CLLB2} &= E_{lb4LB4} + E_{lb16LB32} + E_{lb32LB32} \\ &= ((1.02 \times 10^{-04}) \times (4 \times 10) \times (10^{-08})) \\ &+ ((3.73 \times 10^{-04}) \times (16 \times 30) \times (10^{-08})) \\ &+ ((3.73 \times 10^{-04}) \times (32 \times 80) \times (10^{-08})) = 1.14 \times 10^{-08} J \end{aligned} \tag{6.3}$$

Finally, if the benchmark is executed in a DLB architecture (see Figure 6.3), the first step again is to analyse the data dependencies between loops, but in this case, there is no need to check whether the loops are not incompatible, due to the fact that this kind of loop buffer architecture supports execution of multiple incompatible loops in parallel. This scenario is shown in the case (c) of Figure 6.4. Also in this last case, the program memory and the loop buffer architecture are split in smaller and individual sizes to fit the necessity of each instruction cluster that form the loop buffer architecture. On the one hand, if two loop buffers with the same size are used (*i.e.*, 32 instruction

words), the energy consumption is estimated by Equation 6.4. On the other hand, if the choice is to adapt the size of loop buffers to the loops that are executed over them (*i.e.*, loop buffers of 16 and 32 instruction words), the energy consumption is estimated by Equation 6.5. Based on these results, it is possible to conclude that any improvement in the ILP (*Instruction-Level Parallelism*) of the system brings not only improvements in performance, but also improvements in the energy consumption of the system. The increase in ILP makes easy the adaptation of the sizes of the loop buffers to the sizes of the loops that form the application, because it gives more freedom to combine the execution of the loops that form the application. In this last loop buffer architecture, the control logic is more complex than the controllers of the previous loop buffer architectures, but this control logic considerable reduces the execution time and the overall energy of the application because it allows the parallel execution of loops with different iterators.

$$\begin{aligned} E_{DLB1} = & \quad E_{lb4LB32} + E_{lb16LB32} + E_{lb32LB32} \\ = & \quad ((3.73 \times 10^{-04}) \times (4 \times 10) \times (10^{-08})) \\ + & \quad ((3.73 \times 10^{-04}) \times (16 \times 30) \times (10^{-08})) \\ + & \quad ((3.73 \times 10^{-04}) \times (32 \times 80) \times (10^{-08})) = 1.15 \times 10^{-08} J \end{aligned} \tag{6.4}$$

$$\begin{aligned} E_{DLB2} = & \quad E_{lb4LB16} + E_{lb16LB16} + E_{lb32LB32} \\ = & \quad ((1.72 \times 10^{-04}) \times (4 \times 10) \times (10^{-08})) \\ + & \quad ((1.72 \times 10^{-04}) \times (16 \times 30) \times (10^{-08})) \\ + & \quad ((3.73 \times 10^{-04}) \times (32 \times 80) \times (10^{-08})) = 1.04 \times 10^{-08} J \end{aligned} \tag{6.5}$$

As seen after comparing the loop buffer architectures, embedded systems designers can face a trade-off between the energy budget of the system and the performance that is required by the application running on the system. As shown in the motivating example of this Subsection, conventional loop buffer architectures (*i.e.*, CELB and CLLB architectures) are not good enough in terms of energy efficiency due to the high overhead that these loop buffer architectures show. Due to this fact, DLB architectures appear as a promising option to improve the energy efficiency of IMOs. Section 6.6 not only presents the systematic analysis of this trade-off for DLB architectures, but clearly demonstrates that the area overhead and the selected memory technology for the loop buffer architecture have to be taken into account in this trade-off.

6.4. Implementation of the DLB Architecture

The details of the proposed architectural implementations of the DLB architecture are presented in this Section. In the proposed architectural enhancements, multiple loops can be executed in parallel, without the overhead/limitations mentioned in Section 6.2. Multiple synchronizable loop controllers enable the execution of multiple loops in parallel as each loop has its own loop controller. However, the logic of the loop controller is simplified and the hardware overhead is minimal as it has to execute only loop code. Figure 6.5 shows the implementation of the generic architecture of this loop buffer architecture. As shown in this Figure, the general implementation of the DLB architecture is composed by a loop buffer controller and a loop buffer memory.

On the one hand, the loop buffer controller is the control logic circuit that is shown in Figure 6.5. The main components of this part of the loop buffer architecture are a local program counter register (Local PC register), a jump register (JMP register), a wait counter (Wait Counter register), and an output logic. The Local PC register stores the program count that is used locally in the loop buffer architecture. The JMP register stores the address where the local program counter has to jump. The Wait Counter register contains the number of cycles that the loop buffer architecture has to wait in order to be synchronised. The output logic is the part of the circuit that, based on the state of the registers that form the loop buffer controller, selects the instruction that has to be fetched to the data-path. On the other hand, while the loop buffer controller is common to the entire set of architectural implementations of the DLB architecture that are proposed in this Chapter, the loop buffer memory can be implemented in different ways depending of the option that is selected. This memory is the block that is labelled *Data* in Figure 6.5.

This Chapter proposes to the embedded systems designer three promising options for the implementation of the DLB architecture. Each one of these options is a non-overlapping complementary choice that is most suited for a distinct partition of the design space that is related to the implementation of the DLB architecture. Besides, each option is clearly optimal for a specific pattern of application code based on the characteristics of the application like the number of different instructions, loop body sizes, regularity, and width of instructions. In order to demonstrate that these different options to implement the DLB architecture cover the entire design space of this kind of IMO, every option is explained in detail in each one of the following Subsections.

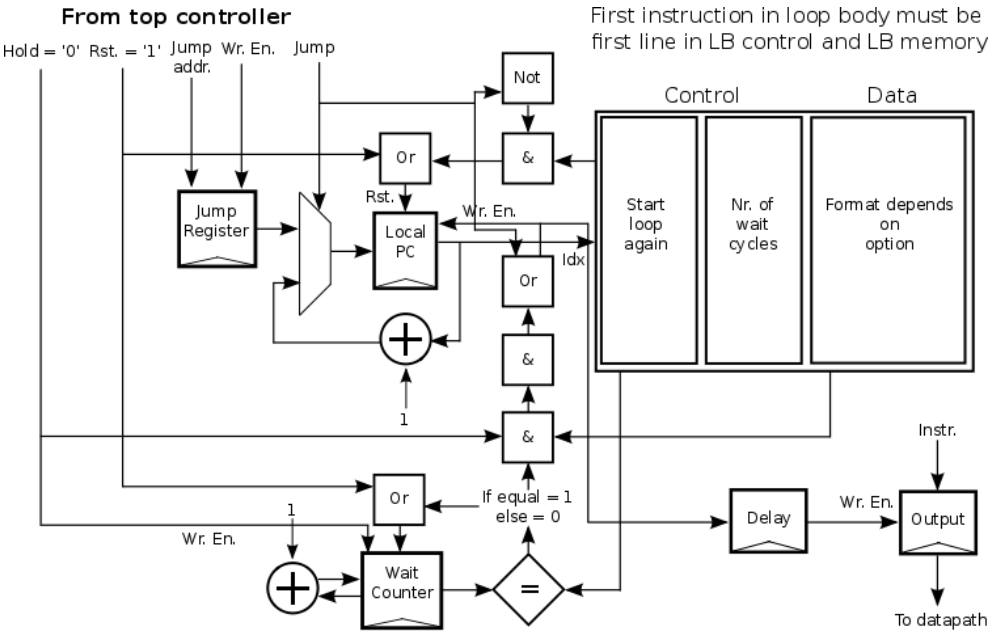


Figure 6.5: Control logic and table of the general format of a DLB architecture.

6.4.1. DLB Architecture - OPTION 1

The code that forms the bodies of the loops that are contained in an application can be sequential or non-sequential. The option that is presented in this Subsection for the implementation of the DLB architecture is most suited if the instructions are fetched from the loop buffer architecture in a sequential order. An example of this kind of pattern can be found in functional units that require instructions in a sequential order like the AG (*Address Generation Unit*).

As shown in Figure 6.6, the LO instruction memory is replaced by a counter. This counter is the element of the logic circuit that controls which instruction is executed in the functional unit associated to the loop buffer architecture. Due to the fact that the counter encodes the instructions based on their positions in the sequential order of fetching, the size of the instructions does not affect how this counter is implemented. In order to control this counter, several fields are required in the control table of the loop buffer architecture. These fields control the start of the execution of the loop, the number of wait cycles, the initialisation of the counter, and the run-time state of the counter. Besides, if the loop buffer architecture needs to run special instructions, an optional small loop buffer memory can be included and controlled by an extra field that can be added to the control table of the loop buffer architecture.

This loop buffer architecture is specifically intended for embedded system designs that exhibit an instruction fetch sequence which is regular and incremental. The advantage of this option is that it has a small overall cost because it does not increase the storage in the L0 memory and it requires only very simple control logic. Moreover, this loop buffer architecture option can deal with narrow and wide instructions with no modification.

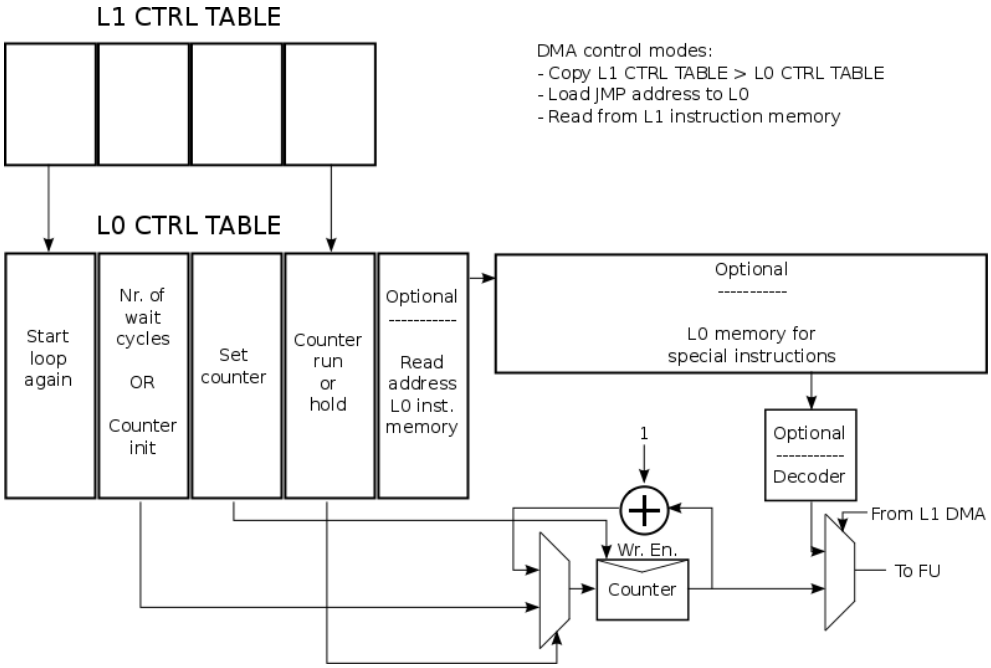


Figure 6.6: DLB architecture - OPTION 1.

6.4.2. DLB Architecture - OPTION 2

The option in Subsection 6.4.1 clearly has restrictions. Not every code that forms the body of a loop is regular and incremental in nature. The alternative option presented in this Subsection is most suited when the instructions are fetched from the loop buffer architecture in a non-incremental order. First, this Subsection discusses the case when the instruction word/fields are relatively narrow. The other case is discussed further in Subsection 6.4.2.

As shown in Figure 6.7, the access of the loop buffer architecture is direct because instead of having a loop buffer memory, the instructions for the functional unit that is associated to the loop buffer memory are directly coded in the control table of the loop buffer architecture, so additional storage space has to be allocated for this. In this option, the control table is the element

that controls which instruction is executed in the functional unit associated to the loop buffer architecture. Due to the fact that the instructions are encoded in the control table of the loop buffer architecture, the size of the instructions affects the energy consumption of the loop buffer architecture. Therefore, this option of implementation of the DLB architecture is most suited if the instructions that are stored in the control table are narrow. As can be seen from Figure 6.7, the control of this implementation is simpler than the implementation that is presented in Subsection 6.4.1. Indeed, only two fields are required in the control table to manage the start of the execution of the loop and the number of wait cycles.

The advantage of this option is that the loop buffer architecture can reduce greatly the energy consumption of the IMO if the number of bits that are used to encode the instructions is small. With this option, embedded systems designers do not have to take into account the order of the instruction fetch that the functional unit associated to the loop buffer architecture follows.

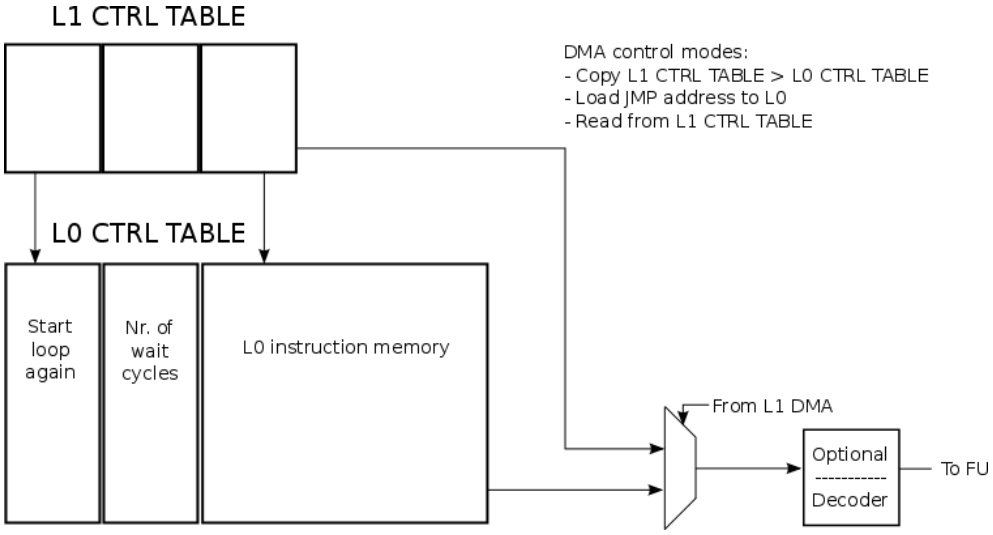


Figure 6.7: DLB architecture - OPTION 2.

6.4.3. DLB Architecture - OPTION 3

Section 6.4.2 presents an option of implementation of the DLB architecture, which principal requirement for energy savings is that the instructions that are used in the loop buffer architecture are narrow. The option presented in this Subsection covers the remaining part of the design space of the DLB architecture. It is most suited if the instructions that are fetched from the loop buffer architecture are non-regularly accessed and wide but when only

few distinct instructions are present. In this way, the entire design space of this IMO is completely covered.

As shown in Figure 6.8, this implementation of DLB architecture has a loop buffer memory therefore the instructions for the functional unit that is associated to the loop buffer architecture are stored in its own memories. In this option, the control table is also the element of the logic circuit that controls which instruction is executed in the functional unit associated to the loop buffer architecture. However, due to the fact that the instructions are stored in own loop buffer memories, the control table increases its complexity in order to control the flow of accesses to the loop buffer memories. This is the reason why the control table of the option presented in this Subsection has a field which purpose is to store the address for indirect indexing of the instruction that is stored in the loop buffer memory. Based on these characteristics, this option of implementation of the DLB architecture is most suited if the instructions that are used in the loop buffer architecture are wide, because this implementation avoids to store the instructions in the control table. From Figure 6.8, it is possible to see that the control of this implementation is simpler than the implementation that is presented in Subsection 6.4.1, but more complicated than the implementation that is presented in Subsection 6.4.2. In the implementation that is presented in this Subsection, three fields are required in the control table of the loop buffer architecture. These fields control the start of the execution of the loop, the number of wait cycles, and the address of the loop buffer memory in which the instruction is stored.

In order to obtain energy savings, this option has as requirement that the instructions that are used in the loop buffer architecture are wide. However, two possible scenarios can appear in the way the body of a loop is formed. On the one hand, the body of the loop can be formed by many different instructions. In this case, the control table stores the address of the loop buffer memory in which the instructions are stored. In this way, the loop buffer memory is accessed by indirect indexing. On the other hand, the body of the loop can be formed by few different instructions. In this case, the field that stores the address of the loop buffer memory in which the instruction is stored can be removed. In this way, the loop buffer memory is accessed by direct indexing, which reduces even more the energy consumption of the loop buffer architecture.

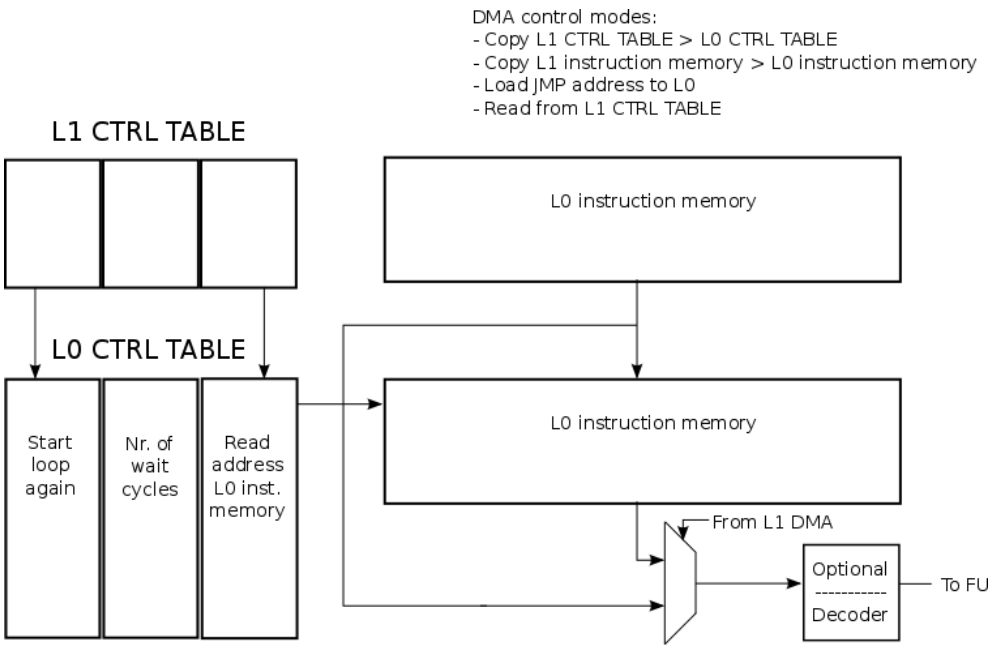


Figure 6.8: DLB architecture - OPTION 3.

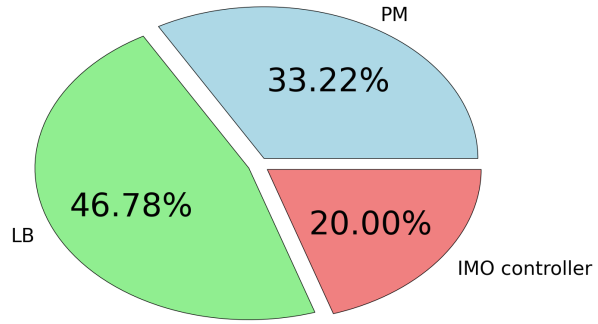
6.5. Experimental Framework

For the case studies that are presented in this Chapter, an experimental framework is built, which is composed of a DMH (*Data Memory Hierarchy*), an IMO, a processor architecture, a loop buffer architecture, and an I/O interface. The energy simulations, that are shown in this Chapter, are performed using the high-level energy estimation and exploration tool that is described in Chapter 3. For a given application and compiler, this tool explores different loop buffer architectures and configurations that can compose the IMO. As shown in Section 3.3, in order to correctly model the loop buffer architectures that are presented in Section 6.2, this tool uses energy models of each one of the components that form the IMO. Based on these models, the characteristics of these components are adapted to the requirements of the processor architecture and the application that is executed in the embedded system. As shown in Section 3.3, this tool requires three inputs: the store access history report of the application, the requirements of the embedded system, and the cycle-level energy models of the memory instances that can compose the IMO. As outcome of the processing of its inputs, this high-level energy estimation and exploration tool provides the energy profiles of the components that form each one of the representative IMOs.

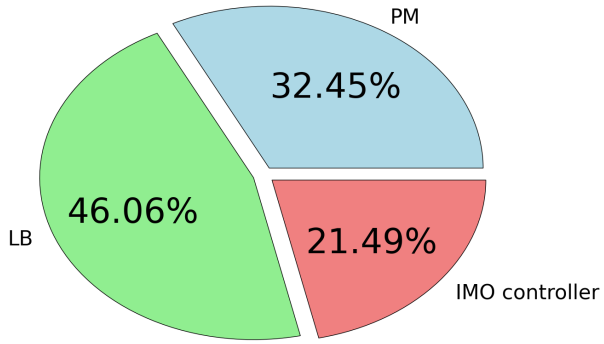
The loop buffer architecture consists of a loop buffer memory and a loop buffer controller. This implementation of the loop buffer memory is based on a set of banks, in which each bank can be configured to fit the desired size and number of the instruction words. The loop buffer controller is the component that monitors the state of the loop buffer memory inside of the IMO. The energy models of the loop buffer memory and loop buffer controller can be obtained in two ways. Either from the data sheets of the commercial memories that the designer wants to use (e.g., a SRAM (*Static Random Access Memory*)-based memory, register-based memory, FF-based memory), or by creating the energy models of the memory instances from post-layout back annotated energy simulations (e.g., loop buffer controller, FFs). In this last option, RTL (*Register-Transfer Level*) simulations have to be performed to produce accurate energy waveforms of the instance under modelling with some user-specified training VCD (*Value Change Dump*) files. In order to cover the whole design space of an instance, variations in depth, width, and technology that is used in the implementation of the memory are done. The evaluation that is presented in this Chapter is performed using TSMC (*Taiwan Semiconductor Manufacturing Company*) 90nm LP (*Low Power*) libraries and commercial memories.

6.6. Experimental Results

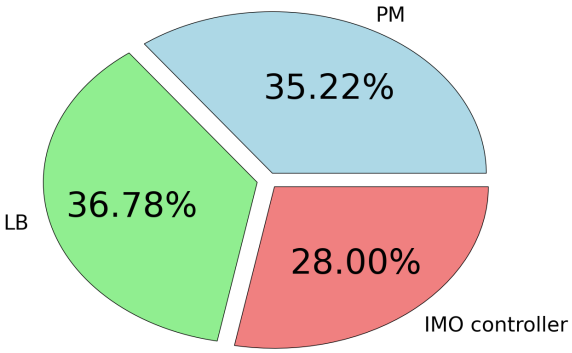
Firstly, this Section shows based on experimental results how DLB architectures sacrifice a portion of energy consumption in the control logic of the IMO in order to save energy in the overall IMO. The trade-off between the energy consumption of the control logic of the IMO and the possible energy savings in the IMO is shown in Subsection 6.6.1. Secondly, the options for the implementation of the DLB architecture that are proposed in this Chapter are analysed with several case studies. The high-level trade-off analysis of the complete energy design space exploration of the DLB architecture, that it is composed by the non-overlapping and complementary implementations that are presented in this Chapter, requires benchmarks with different patterns in their constitution and execution. On the one hand, Subsection 6.6.2 describes the synthetic benchmarks that have been developed to show the trends in energy consumption of each one of the different architectural implementations that are proposed in Section 6.4. On the other hand, Subsection 6.6.3 presents the evaluation and analysis of the proposed implementations based on real-life embedded applications to show their trend in energy consumption.



(a) CELB architecture.



(b) CLLB architecture.



(c) DLB architecture.

Figure 6.9: Energy breakdown for different LB architectures running the benchmark *AES NON-OPTIMISED*.

6.6.1. Comparison to Conventional Solutions

The benchmark *AES-NON OPTIMISED* that is presented in Table 6.2 is used in this Subsection. Figure 6.9 shows how for this benchmark the distribution of the energy consumption in the IMO changes from one representative architecture to another. If the CELB architecture and the CLLB architecture are compared, it is possible to see that the energy related to the loop buffer controller is increased in the last loop buffer architecture, due to the fact that the controller has to control the activation or deactivation of more components. Also, it is possible to appreciate that the consumption of the loop buffer is reduced. This fact is related to the reduction in the dynamic energy consumption that is caused by the use of smaller memory instances. If the DLB architecture is analised, it is possible to appreciate that the percentage of the energy consumption that is related to the loop buffer controller of this loop buffer architecture is higher than the loop buffer controllers of the previous loop buffer architectures. This is due to the fact that the loop buffer controller of the DLB architecture is more complex than the previous loop buffer architectures. However, as it is possible to see from this Figure, the efficient management that is performed by the loop buffer controller leads to higher reductions in the loop buffer architecture and, as a consequence in the overall energy of the IMO. It should be noted that the absolute value of the energy consumption of the program memory is not a constant in all these IMOs due to two facts. Firstly, the number of cycles that the application requires for its execution over these loop buffer architectures changes. Secondly, the size and width of the memory changes, as well as its composition when it is banked. The variation in the size and/or width of the memories that forms the loop buffer memory affects more the CELB architecture than the CLLB architecture or the DLB architecture, because the efficient management, that the control logic of these last loop buffer architectures have for the activation or de-activation of the memory instances, compensates the possible overhead that can be created by the variation of the memory instances that form the loop buffer memory.

6.6.2. Synthetic Benchmarks

The complete energy design space exploration of possible implementations of the DLB architecture is based on the architectural models presented in Section 6.4. In order to see the potential energy savings of each one of these architectural implementations, benchmarks with specific patterns have to be used to clearly show the special case in which each option is optimal. For that purpose, specific synthetic benchmarks are implemented. Table 6.2 presents these specific synthetic benchmarks. These synthetic benchmarks mimic loops that one can find in real-life embedded applications. Every loop

Table 6.2: Synthetic and real-life embedded applications used as benchmarks.

Synthetic Benchmark (See Section A.6.2.2)	Cycles	Issue slots	Bits per instruction	LB size [Instructions]	Loop code [%]	NOP instructions [%]
SB 1	3,050	2	16	32	96.88	0.07
SB 2	3,050	2	16	32	96.88	0.07
SB 3	3,050	2	32	32	96.88	0.07
Real-life Benchmark [Reference]	**Cycles**	**Issue slots**	**Bits per instruction**	**LB size [Instructions]**	**Loop code [%]**	**NOP instructions [%]**
AES NON-OPTIMISED [TSH+10] (See Section C.2)	707,052	1	16	64	1.68	8.03
AES OPTIMISED [TSH+10] (See Section C.3)	3,347	1	16	32	77.44	0.09
BIO-IMAGING [PFH+12] (See Section C.4)	334,071	4	80	64	98.01	26.70
CWT NON-OPTIMISED [YKH+09] (See Section C.5)	274,464	1	16	32	6.61	0.48
CWT OPTIMISED [YKH+09] (See Section C.6)	102,827	1	20	64	6.36	2.50
DWT NON-OPTIMISED [DS98] (See Section C.7)	758,216	2	16	518	65.70	25.19
DWT OPTIMISED [DS98] (See Section C.8)	317,739	2	32	4	1.89	1.73
MRFA [QJ04] (See Section C.9)	177,170	2	32	64	19.13	17.01

that is included in the synthetic benchmarks is characterised based on two parameters: the size of the loop body and the number of loop iterations. The ranges of loop body sizes in the synthetic benchmarks are shown in Figure 6.4. All the synthetic benchmarks that are used in this Subsection are based on the example shown in Figure 6.4. Therefore, every benchmark is composed of three loops of 4, 16, and 32 instruction words respectively. In order to control with sufficient accuracy the sizes of the loop bodies, the synthetic benchmarks are implemented in assembly code. The instructions and their operands that compose each loop are randomised depending on the synthetic benchmark that it is implemented. This is different from reality where some correlation is present in these instruction bits, but for the purpose of the experiment in this Subsection, these correlations are not that relevant, because they have low impact on the energy consumption of the loop buffer architecture.

The three synthetic benchmarks that are used in this Subsection are described in the following bullets:

SB 1 This synthetic benchmark is intended to show when the option number 1 is the most energy efficient implementation of the possible options to implement the DLB architecture shown in Subsection 6.4.1. The option number 1 is specifically intended for embedded system designs that exhibit an instruction fetch sequence which is regular and incremental. Therefore, based on this premise, the *SB 1* is composed of loops that present instructions that are fetched from the loop buffer architecture in a sequential order. The rest of the characteristics of this synthetic benchmark are summarised in Table 6.2.

SB 2 The advantage of the option number 2 is that the loop buffer architecture can greatly reduce the energy consumption of the IMO, if the number of bits that are used to encode the instructions is small. Due to the fact that in this option embedded systems designers do not have to take into account the order of the instruction fetch that the functional unit associated to the loop buffer architecture follows, *SB 2* is implemented with the same number of bits per instruction as *SB 1*. However, in this case, the loops that form this benchmark present non-regular sequential order when instructions are fetched from the loop buffer architecture. The instructions that form each loop of this benchmark are randomised with a 50 % probability of change in relation with the pattern of instructions that is presented by the synthetic benchmark *SB 1*.

SB 3 This last synthetic benchmark is intended to show the case in which the more energy efficient option is the number 3 from the ones that are presented in Subsection 6.4.3. In this synthetic benchmark, the loops that form the synthetic benchmark not only present non-regular sequential order of instruction fetch, but also the number of bits

per instruction is increased (wide instruction word) compared to the synthetic benchmarks *SB 1* and *SB 2*.

Figure 6.10 presents the normalised energy consumption of each one of the proposed implementations for the DLB architecture when the synthetic benchmarks that are described in the previous paragraph are running on them. In this Figure, the proposed implementations are compared with the CELB architecture. As can be seen, when *SB 1* is used, the most efficient option is indeed the number 1. As *SB 1* is a synthetic benchmark with a small number of bits per instruction, the implementation option number 2 is better than the implementation option number 3, because option number 3 improves its energy savings when the number of bits per instruction is increased. From the results of *SB 2*, it is possible to see that, in this case, the best implementation option is not the number 1, but the number 2. Besides, the implementation option number 1 is better than the implementation option number 3, because despite the penalty of the non-regular sequence of instruction fetch, the number of bits per instruction introduces a big overhead in the energy consumption. As it is mentioned before, the synthetic benchmark *SB 3* is the same as the synthetic benchmark *SB 2*, but the first benchmarks has more bits per instruction. Due to this fact, when the benchmark *SB 3* is running over the different options, only the implementation option number 3 is energy efficient. The low overhead in control logic that presents the implementation option number 1 compared to the implementation option number 2 makes the first option more energy efficient when the synthetic benchmark *SB 3* is used. From the point of view of the implementation option number 1, the best synthetic benchmark is *SB 1*. However, it is possible to see that there is not a big difference between the execution of *SB 2* and *SB 3*, because for this implementation option, these benchmarks present the same execution pattern. From the point of view of the implementation option number 2, the best synthetic benchmark is either *SB 1* or *SB 2*, because both have a small number of bits per instruction. *SB 3* is not good for this implementation option, because this synthetic benchmark has a bigger number of bits per instruction compared to *SB 1* and *SB 2*. From the point of view of the implementation option number 3, only the synthetic benchmark *SB 3* is good due to the application characteristic of the number of bits per instruction. The regular sequence or not in the fetch instruction stage is not important for this implementation option, as it is possible to see due to the small difference between the execution of the synthetic benchmark *SB 1* and *SB 2*.

The conclusion of this Subsection is that there are specific patterns, which can be found in the application code, that help to embedded systems designers to choose the optimal implementation of the DLB architecture from the energy consumption point of view and this fact will be exploited in following Sections.

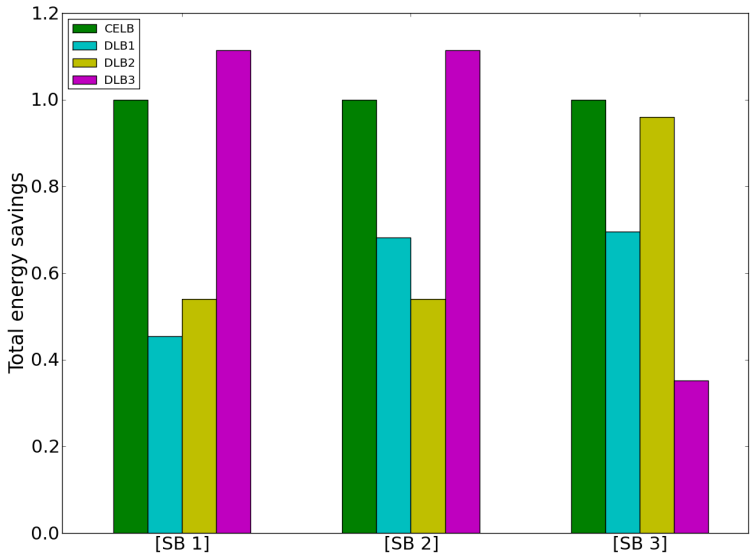


Figure 6.10: Normalised energy consumption in the DLB architectures based on synthetic benchmarks.

6.6.3. Real Benchmarks

Figure 6.11 shows that it is possible to increase from 6 % to 22 % the overall energy savings for the considered benchmarks of the IMO from the CLLB architectures to the DLB architectures, for that an increase of approximately 6.5 % of the energy consumption of the control logic is required.

In order to corroborate the benefits and disadvantages of each one of the options that are proposed in this Chapter for the implementation of the DLB architecture, real-life embedded applications are used as benchmarks to complete the high-level analysis proposed in this Chapter. Table 6.2 presents the characteristics of these real-life benchmarks. The selected benchmarks are prime examples not only of all application domains that are loop dominated, exhibit sufficient opportunity for DLP (*Data-Level Parallelism*) and/or ILP (*Instruction-Level Parallelism*), comprise signals with multiple word-lengths, and require a relatively limited number of variable multiplications, but also of more general-purpose applications domains that can be found in the area of wireless base-band signal processing, multimedia signal processing, or different types of sensor signal processing. This selection of the benchmarks was performed with the goal of making the analysis that is presented in this Chapter generic enough, in order to be applicable to all loop-dominated application domains in embedded systems.

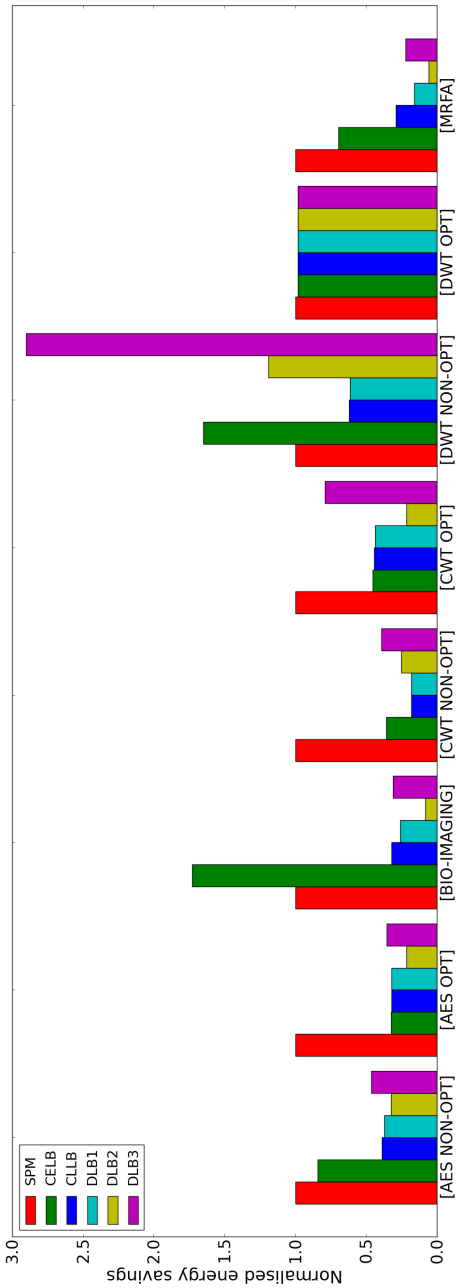


Figure 6.11: Normalised energy savings in different IMOs.

Figure 6.11 presents the normalised energy savings in logarithmic scale of each one of the proposed implementations for the DLB architecture when the real-life benchmarks are running on them. As can be seen in this Figure, the implementation option number 1 is the best choice for many of the real-life benchmarks, but not for the benchmarks *CWT NON-OPTIMISE* and *DWT NON-OPTIMISE*. The fact that both benchmarks are not optimised by an embedded systems designer means that the good results of the implementation option number 1 are related with the regular sequence of fetch instructions that appears in general real-life benchmarks. The degree of difference between the implementation option number 1 and the other two options depends of the regular sequence of the instructions when they are fetched from the loop buffer architecture. This fact is appreciated between optimised implementations of the real-life benchmarks and non-optimise versions of them. From the point of view of the implementation option number 2, benchmark *DWT OPTIMISE* clearly takes advantage of this implementation option of the DLB architecture. This is due to the small size and number of instructions that have to be stored in the loop buffer control table of the loop buffer architecture. From the point of view of the implementation option number 3, the real-life benchmarks *CWT NON-OPTIMISE* and *DWT NON-OPTIMISE* are the benchmarks that take advantage of the energy savings that offers this implementation option. These benchmarks present non-regular sequence of instruction fetch. Besides, the bits per instruction and the number of instructions that have to be stored in these real-life benchmarks are both high. It is possible to see several anomalies along the experimental results that are presented by these real-life benchmarks. The first anomaly is in the benchmark *BIO-IMAGING*. This anomaly is related to the big size of the memory instance that is required by the CELB architecture to run this benchmark.

For this real-life benchmark, during loop code parts of the application, a single memory instance has to store 64 instructions of 80 bits each one of them. This big memory instance is completely active when only one single address is accessed. This fact increases a lot the energy consumption of the overall IMO. The rest of the loop buffer architectures have more memory instances that are smaller and can be active or not at the same time. In the case of the benchmark *DWT NON-OPTIMISE*, it is the same problem than the one related to the benchmark *BIO-IMAGING*. However, in this case, the memory instances are bigger as shown in Table 6.2. This problem is bigger in the DLB implementation option number 2 than in the CELB architecture, because in the implementation option number 2 the memory instance contains not only the instruction code, but also the bits that control the loop buffer architecture. In the case, of the CELB architecture, the memory instance only has to store the instructions in a single big memory instance.

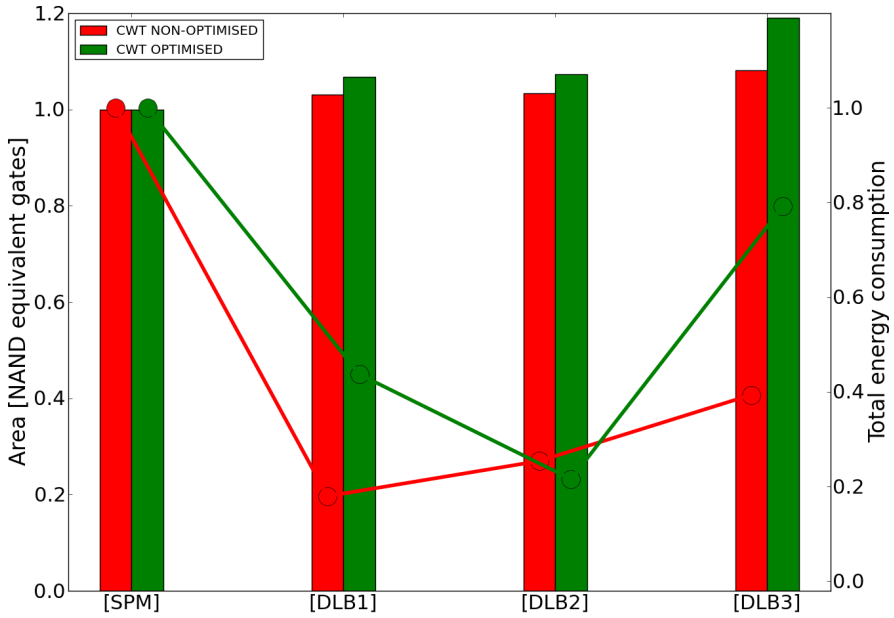


Figure 6.12: Normalised energy consumption (represented by lines) vs. occupancy (represented by solid bars) in different IMOs.

Figure 6.12, based on area-energy Pareto curves, shows the overhead in the memory area that is required to achieve the energy savings of two of the benchmarks of Table 6.2. As shown in this Figure, the area penalty, that the embedded systems designer has to assume in order to achieve further energy savings, is relatively small (10 % in average). But this still shows an interesting trade-off to be decided based on the overall design context. The application characteristics as well as the system requirements have to be analysed to make the decision in this trade-off.

From the results that are obtained along this Section, it is possible to conclude that each option is clearly optimal for a specific pattern of application code. Due to this fact, the optimal energy efficient IMO has to combine these non-overlapping and complementary implementation options to cover the distinct partitions of the design space of the DLB architecture. If the embedded system is designed only for one single specific application, the embedded systems designer can fix the parameters to achieve the optimal specific configuration that creates the most efficient implementation of the IMO for the given application code. However, if there are a set of applications that the user wants

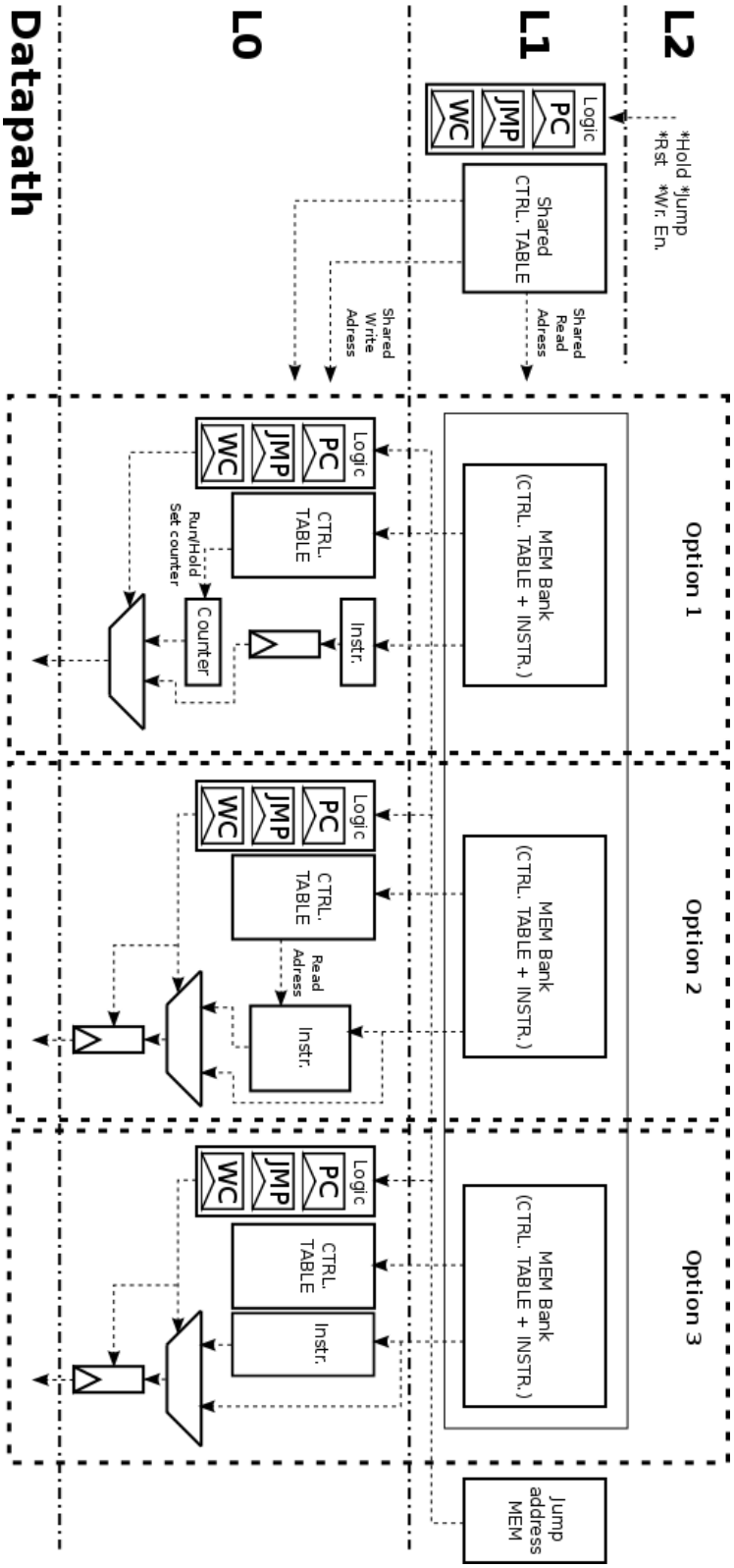


Figure 6.13: Example of a system with the 3 options of implementation of the DLB architecture.

to run on the embedded system, in this case, this system has to use dynamic reconfiguration based on dynamic profiling in order to be adapted to the all the applications that the user wants to run on the embedded system. In this way, the system uses the most efficient implementation of the DLB architecture for each specific piece of the application code that match one of the patterns shown in Subsection 6.6.2. Figure 6.13 shows an example of an embedded system with the non-overlapping options of implementation of the DLB architecture that are proposed in this Chapter. If due to issues in the design of the embedded system, only one option of the proposed implementations of the DLB architecture has to be chosen, the guideline to use each option based on the characteristics of the application as number of different instructions, loop body sizes, regularity, and width of instructions is:

- **OPTION 1.** The application code presents regular sequence in the fetch of instructions from the loop buffer architecture. The number of bits per instruction in the codification of the application does not affect the energy efficiency of this implementation option of the DLB architecture.
- **OPTION 2.** The application code presents instructions that have a small number of bits in their codification. The sequence in the fetch instruction does not affect the energy efficiency of this implementation option of the DLB architecture.
- **OPTION 3.** The application code presents instructions that have a big number of bits in their codification. The sequence in the fetch instruction does not affect the energy efficiency of this implementation option of the DLB architecture.

6.7. Conclusions

Instruction memory organisations have been pointed out as one of the major sources of energy consumption in embedded systems. As embedded systems are characterised by restrictive resources and a low-energy budget, any enhancement that is introduced in this component of the system becomes crucial in order to decrease this energy bottleneck. Distributed loop buffer architectures with incompatible loop-nest organisations are promising options to improve the energy efficiency of the instruction memory organisations. These distributed and scalable implementations work like multi-threaded platforms allowing the execution of incompatible loops in parallel with minimal hardware overhead. This Chapter proposes and analyses three options, where each one of these options is a non-overlapping complementary choice that is most suited for a distinct partition of the design space related to the distributed loop buffer architecture implementation. The high level trade-off analysis of the proposed implementations is crucial to present the correct

process design that an embedded systems designer has to follow in order to have an efficient implementation of a distributed loop buffer architecture for a certain application, helping the embedded systems designer to take decisions in early stages of the design, which can dramatically affect the energy consumption of the embedded system. Results show that, with an increase of about 6.5 % in the energy consumption of the control logic that exists in the instruction memory organisation, the overall energy consumption of the instruction memory organisation can be reduced by 6 % to 22 %, when distributed loop buffer architectures with incompatible loop-nest organisations are used instead of clustered loop buffer architectures with shared loop-nest organisations architectures.

In the next chapter...

the reader will find a synthesis of the conclusions that are derived from the research that is presented in this Ph.D. thesis, but also the major contributions to the state-of-the-art for the development of loop buffer architectures for instruction memory organisation in embedded systems. Besides, this Chapter also includes a summary of the open research lines and the future research directions.

Chapter 7

Conclusions and Future Work

"The best structure will not guarantee results and performance. But the wrong structure is a guarantee of non-performance."

— Peter Drucker.

This Ph.D. thesis has introduced the study, analysis, proposal, implementation, and evaluation of low-energy optimisation techniques that can be used in instruction memory organisations of embedded systems. In this Chapter, a synthesis of the conclusions derived from the research that has been proposed in this Ph.D. thesis is presented. This Chapter not only summaries the major contributions of this Ph.D. thesis to the state-of-the-art for the development of loop buffer architectures for instruction memory organisations in embedded systems, but also highlights the open research lines and the future research directions that are derived from this Ph.D. thesis.

7.1. Summary

As shown in Section 1.1.1, embedded systems not only have a market size that is about 100 times the desktop market, but also they offer the widest variety of processing power and cost portion of the electronic market. However, despite this variety, every embedded design is constrained by the following issues: the required performance, the optimisation of memory area, and the reduction of energy consumption. Modern embedded applications constrain the design of actual embedded systems. Most of these applications require not only sustained operation for long periods of time, but also to be executed on battery powered systems. Under the constraint of not being mains-connected, the absence of wires to supply a constant source of energy causes that the use of an energy harvesting source [CC08] or an integrated energy supplier [AG09] limits the operation time of these electronic devices. Due to these

facts, the optimisation of the energy consumption is nowadays becoming very crucial in the design of energy-efficient embedded systems, not only due to the increasing demand of battery powered systems, but also due to the need of using less expensive packaging. In order to achieve the low-power constraints that modern embedded applications require, it is crucial not only to decrease the total energy consumption of all the components of the embedded system, but also to have a better distribution of the energy budget throughout the whole embedded system. Therefore, embedded systems designers have to look at the complete system and tackle the energy consumption problem in each component of the system.

Previous research, as can be seen in Section 1.3, has pointed out the instruction memory organisation as one of the major sources of energy consumption in embedded instruction-set processor platforms. As these systems are characterised by restrictive resources and a low-energy budget, any enhancement that is introduced in the instruction memory organisation allows not only to decrease the energy consumption, but also to have a better distribution of the energy budget throughout the embedded system. This Ph.D. thesis has introduced the study, analysis, proposal, implementation, and evaluation of low-energy optimisation techniques that can be used in instruction memory organisations of embedded systems. The next paragraphs present the summaries and the conclusions of each one of the Chapters that form this Ph.D. thesis.

- Chapter 1 has presented the motivation and problem context of the energy consumption of the instruction memory organisation in embedded systems. Besides, this Chapter has provided an overview of the content of this Ph.D. thesis.

- Chapter 2 has provided a complete literature study and an up-to-date picture of the current status of the low-energy techniques that are used in instruction memory organisations, giving to the reader not only a first grasp on the fundamental characteristics and design constraints of various types of instruction memory organisations, but also an outline of their comparative advantages, drawbacks, and trade-offs.

- Chapter 3 has proposed a high-level energy estimation tool that finds the optimised configuration for the total energy consumption of the embedded system, by exploring different configurations of the instruction memory organisation, for both given application and compiler. Apart from the reduction in time and effort to explore the design space of the instruction memory organisation, the proposed tool allows the exploration of the effects that are caused by code transformations and compiler configurations in the total energy consumption of the instruction memory organisation.

- Chapter 4 has presented a high-level trade-off analysis of promising loop buffer schemes for embedded systems, which serves to embedded systems designer as a guideline in the objective to implement an instruction memory organisation with a low-cost energy per task. This Chapter has showed that the possible energy reductions (up to 76 %) depends not only on the total percentage of time which is spent inside the loop code, but also on the size, implementation, and handling conditions of the loop buffer architecture that is used. Moreover, Chapter 4 has proposed a run-time loop buffer architecture that optimises both the dynamic energy consumption and leakage energy consumption of the instruction memory organisation. Based on post-layout simulations, this run-time loop buffer architecture improves the energy consumption of the instruction memory organisation by average of 20 % in comparison with a loop buffer architecture based on a single monolithic memory, and more than 90 % in comparison with instruction memory organisations without loop buffer architectures.

- Chapter 5 has shown how the loop buffer concept is applied in real-life embedded applications that are widely used in biomedical wireless sensor nodes, to show which scheme of loop buffer is more suitable for applications with certain behaviour. The loop buffer architectural organisations that have been analysed in this Chapter are the CELB (*Central Loop Buffer Architecture for Single Processor Organisation*) and the BCLB (*Banked Central Loop Buffer Architecture*). From the experimental evaluation that is presented in this Chapter, gate-level simulations have demonstrated that a trade-off exists between the complexity of the loop buffer architecture and the power benefits of utilising it. Two factors have to be taken into account in order to implement an energy-efficient instruction memory organisation based on a loop buffer architecture: (1) the percentage of the execution time of the application that is related with the execution of the loops included in the application, and (2) the distribution of the execution time percentage, which is related with the execution of the loops, over each one of the loops that forms the application.

- Chapter 6 has proposed and analysed non-overlapping and complementary implementation options for distinct partitions of the design space of the DLB (*Distributed Loop Buffer Architecture with Incompatible Loop-Nest Organisation*). DLB architectures are promising options to improve the energy efficiency of instruction memory organisations. These distributed and scalable implementations work like multi-threaded platforms allowing the execution of incompatible loops in parallel with minimal hardware overhead. Results from this Chapter have shown that, with an increase of about 6.5 % in the energy

consumption of the control logic that exists in the instruction memory organisation, the overall energy consumption of the instruction memory organisation can be reduced by 6 % to 22 %, when a DLB architecture is used instead of a CLLB (*Clustered Loop Buffer Architecture with Shared Loop-Nest Organisation*).

In the next Section, the main contributions of this Ph.D. thesis are summarised.

7.2. Main Contributions

As shown in Section 1.4, the main contributions that this Ph.D. thesis has offered to the research community are:

- The systematic study of existing low-energy optimisation techniques that are used in instruction memory organisations, outlining their comparative advantages, drawbacks, and trade-offs.
- The use of post-layout simulations to evaluate the energy impact of the loop buffer architectures in an experimental evaluation with a systematic method in order to have an accurate estimation of parasitics and switching activity.
- The development of a high-level energy estimation tool that, for a given application and compiler, allows the exploration not only of architectural and compiler configurations, but also of code transformations that are related to the reduction of energy and cost of the instruction memory organisation.
- The evaluation of different loop buffer schemes for certain embedded applications, guiding the embedded systems designer to make the correct decision in the trade-offs that exist between energy budget, required performance, and cost area of the embedded system.
- An implementation of a run-time loop buffer architecture that optimises both the dynamic energy consumption and leakage energy consumption of the instruction memory organisation.
- The proposal and analysis of non-overlapping and complementary implementation options for distinct partitions of the design space that is related to DLB architectures.

Each one of the Chapters that form this Ph.D. thesis addresses the contributions that were previously exposed. In terms of scientific publications, this Ph.D. thesis has generated the following articles in international journals:

HSN+a13 Kim, H., S. Kim, N. V. Helleputte, A. Artes, M. Konijnenburg, J. Huisken, C. V. Hoof, and R. F. Yazicioglu, "A Configurable and Low-Power Mixed Signal SoC for Portable ECG Monitoring Applications", Journal of IEEE Transactions on Biomedical Circuits and Systems: IEEE Computer Society, 2013.

ARJFb13 Artes, A., R. Fasthuber, J. L. Ayala, P. Raghavan, and F. Catthoor, "Design Space Exploration of Loop Buffer Schemes in Embedded Systems", Special Issue of Journal of Systems Architecture on Design Space Exploration of Embedded Systems: Elsevier Amsterdam, 2013.

ARJFa13 Artes, A., R. Fasthuber, J. L. Ayala, P. Raghavan, and F. Catthoor, "Design Space Exploration of Distributed Loop Buffer Architectures with Incompatible Loop-Nest Organisations in Embedded Systems", Journal of Signal Processing Systems: Springer New York, 2013.

AJJFa12 Artes, A., J. L. Ayala, J. Huisken, and F. Catthoor, "Survey of Low-Energy Techniques for Instruction Memory Organisations in Embedded Systems", Journal of Signal Processing Systems: Springer New York, 2012.

A.JFb12 Artes, A., J. L. Ayala, and F. Catthoor, "Power Impact of Loop Buffer Schemes for Biomedical Wireless Sensor Nodes", Journal of MDPI Sensors: MDPI AG, 2012.

, and the following articles in international peer-reviewed conferences:

MMJ+a12 Komalan, M., M. Hartmann, J. I. Gomez, C. Tenllado, A. Artes, and F. Catthoor, "System Level Exploration of Resistive-RAM (ReRAM) based Hybrid Instruction Memory Organization", Memory Architecture and Organization Workshop (MeAOW), 2012.

A.JFa12 Artes, A., J. L. Ayala, and F. Catthoor, "IMOSIM: Exploration Tool for Instruction Memory Organisations based on Accurate Cycle-Level Energy Modelling", IEEE International Conference on Electronics, Circuits, and Systems (ICECS), 2012.

HRS+a11 Kim, H., R. Firat, S. Kim, V. N. Helleputte, A. Artes, M. Konijnenburg, J. Huisken, J. Penders, and V. C. Hoof, "A Configurable and Low-Power Mixed Signal SoC for Portable ECG Monitoring Applications", Symposium on VLSI Technology and Circuits, 2011.

AJA+a11 Artes, A., J. L. Ayala, V. A. Sathanur, J. Huisken, and F. Catthoor, "Run-time Self-tuning Banked Loop Buffer Architecture for Power Optimization of Dynamic Workload Applications", IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SOC), 2011.

A. Aa10 Artes, A., "Power Consumption on Loop Buffer based Instruction Memory Organizations", STW.ICT Conference on Research in Information and Communication Technology, 2010.

AFM+a09 Artes, A., F. Duarte, M. Ashouei, J. Huisken, J. L. Ayala, D. Atienza, and F. Catthoor, "Energy Efficiency using Loop Buffer based Instruction Memory Organization", International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA), 2009.

7.3. Future Research Directions

The research that is presented in this Ph.D. thesis has identified those factors that strongly impact the energy consumption of the instruction memory organisation in embedded systems. Besides, this Ph.D. thesis has brought some ideas to efficiently manage the energy consumption of this component of the embedded system, guiding embedded systems designers to make the correct decision in the trade-offs that exist between energy budget, required performance, and cost area of the embedded system. However, some interesting points of future research have emerged during the evolution of this work. Some of them appeared when implementing the designed approaches, while many others have shown up by the current evolution of the trend on the design of embedded systems.

The following paragraphs propose future research directions and improvements to the work presented in this dissertation:

- the high-level energy estimation and exploration tool that is proposed in Chapter 3 can be extended in order to build a simulator that incorporates not only the improvements of the instruction memory organisation, but

also the improvements of other components of the embedded system. A simulation environment can be created to analyse in two ways the different architectural improvements that can be proposed. Firstly, a proposed improvement in one of the components of the embedded system can be analysed in isolation at a given time. Secondly, a whole set of improvements related to each one of the components can be analysed together in the complete embedded platform.

- This Ph.D. thesis has proposed loop buffer architectures that are managed through control logic that is implemented in hardware. Therefore, it would be interesting to analyse the benefits of the efficient control, that can be done by software, in the management of loop buffers through the compiler. In this scenario, the compiler would be responsible for mapping the appropriate parts of the application into the loop buffers, activating them only when they are required. Two aspects should be noted. Firstly, a design that is based on a simpler and general-purpose processor has less hardware in its critical path. Hence, higher processor speed-ups could be achieved at lower area overhead. Although the overall energy dissipation of the system will go up, this overhead is expected to be relatively small. Secondly, hardware components have a view of the application that is based only on its behaviour at run-time. In contrast, compilers have a wider view of the application, because after profiling, it is possible to predict at compile-time the behaviour of the application for a specific input. Due to this advantage, compilers can perform global optimisations in a way that the application can be executed more efficiently.

- In the same way than a CELB architecture has been fabricated and introduced in a configurable and low-power mixed signal SoC for portable ECG monitoring applications [KYS+11], the rest of the loop buffer architectures that have been proposed and analysed in this Ph.D. thesis should be fabricated and introduced in a real implementation of an embedded system. This will allow to corroborate the experimental results that were obtained from post-layout simulations, and which have been presented in this Ph.D. thesis.

- This Ph.D. thesis is focused on the study, analysis, proposal, implementation, and evaluation of low-energy optimisation techniques that can be used in the instruction memory organisations of embedded systems. Therefore, it would be interesting to see the behaviour of the same low-energy optimisation techniques, but in this case, applied in other kind of systems (*e.g.*, desktop systems, server systems).

Appendix A

Resumen en Español

"El idioma -el castellano, el español- llega a ser para nosotros como un licor que paladeamos, y del cual no podemos ya prescindir. Prescindir en el ensayo, en la busca de todos sus escondrijos, de todas sus posibilidades, de todas sus puridades. Ya somos, con tanto beber de este licor, beodos del idioma."

— José Augusto Trinidad Martínez Ruiz (Azorín).

En cumplimiento del Artículo 10 de la normativa de la *Universidad Complutense de Madrid*, de desarrollo del R. D. 99/2011, que regula los estudios de Doctorado (BOUC de 21 de diciembre de 2012), se presenta a continuación un resumen en español de la presente tesis doctoral que incluye la introducción, los objetivos, las principales aportaciones, y las conclusiones del trabajo realizado.

A.1. Introducción

A.1.1. Motivación y Contexto

La rápida mejora que la informática ha sufrido en las últimas décadas se debe, tanto a los avances en la tecnología utilizada para construir los sistemas, como a la innovación en el diseño de estos. A pesar de que a lo largo del tiempo los avances tecnológicos han sido bastante estables, los avances derivados de las mejoras arquitecturales han sido menos consistentes. Como se puede ver en la Figura A.1, desde 2002, la mejora del rendimiento del procesador se ha reducido a un 20 % por año. Esto se debe a los límites impuestos por la máxima disipación de potencia permitida, el poco paralelismo restante para explotar eficientemente, y las latencias de las memorias. Debido a estos inconvenientes, se hace crucial la introducción de técnicas hardware en el diseño de sistemas empotrados, con el fin de ayudar al diseñador a diseñar este tipo de sistemas.

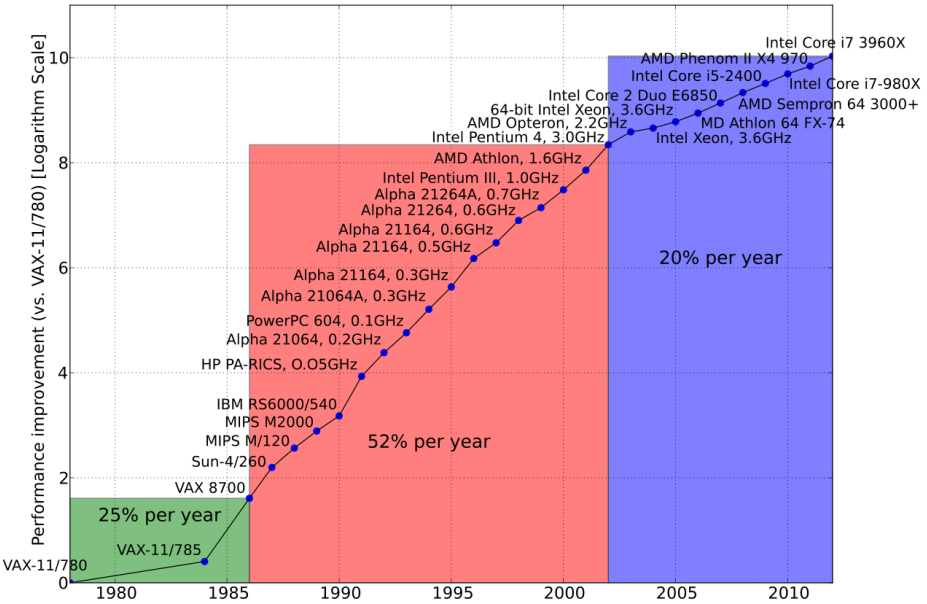


Figura A.1: Crecimiento en el rendimiento del procesador desde mediados de la década de 1980 a 1989.

Esta tesis doctoral se centra en los sistemas empotrados. Estos sistemas son el grupo que más rápido crecimiento ha sufrido y que más amplia variedad de potencia de procesamiento y coste posee dentro del mercado informático. Los sistemas empotrados poseen características especiales. En primer lugar, estos sistemas combinan software y hardware con el fin de ejecutar un conjunto fijo y específico de aplicaciones. Estas aplicaciones son muy diferentes entre sí, ya que van desde las relacionadas con dispositivos de consumo multimedia a las de sistemas de control industrial. En segundo lugar, los sistemas empotrados se caracterizan por poseer no sólo recursos limitados, sino también un consumo bajo de energía, lo que limita su tiempo de funcionamiento. En tercer lugar, a fin de ser fiables y predecibles, los sistemas empotrados proporcionan altas capacidades de cálculo mientras satisfacen las variadas restricciones de tiempo que son impuestas por la aplicación que ejecutan. La combinación de todos estos requisitos hace que el diseño de estos sistemas se convierta en un gran desafío para los diseñadores de sistemas empotrados.

El problema de la limitación de la memoria (*memory wall*) se vuelve aún más complicado en el caso de los sistemas empotrados, donde los diseñadores no sólo deben tener en cuenta el rendimiento, sino también el consumo de energía. Trabajos como [HP07] y [VM07] han demostrado que la OMI (*Organización de la Memoria de Instrucciones*) y la JMD (*Jerarquía de la Memoria de Datos*) poseen porciones del área y del consumo de energía del sistema que

no son insignificantes. De hecho, ambas arquitecturas de memoria representan hoy en día hasta el 40 %–60 % del presupuesto total de energía del sistema empotrado [CRL+10].

Debido al hecho de que esta tesis doctoral se encuentra centrada en los sistemas empotrados, se han usado como *benchmarks* aplicaciones empotradas reales del subdominio específico de los nodos de sensores inalámbricos, a fin de mostrar, analizar y corroborar las ventajas y desventajas de cada uno de los conceptos en los que esta tesis doctoral se encuentra basada. Los *benchmarks* seleccionados se encuentran descritos en el Apéndice C.

A.1.2. Formulación del Problema

A diferencia del diseño de la jerarquía de memoria de los sistemas de propósito general, la cual se encuentra centrada en el rendimiento del sistema, el diseño de la jerarquía de memoria de los sistemas empotrados posee objetivos más diversos donde el área, el rendimiento, el ancho de banda de comunicación, y el consumo de energía se encuentran incluidos con el mismo nivel de importancia. La Figura A.2 muestra el desglose de potencia de un sistema empotrado real, el cual se encuentra basado en un sistema SoC configurable y de bajo consumo para aplicaciones de monitorización de señales de ECG (ver Sección C.6). En esta Figura A.2, se puede ver que tanto la OMI (en inglés, IMO) como JMD (en inglés, DMH) dominan fuertemente el consumo de energía del sistema empotrado. Este caso de estudio demuestra que optimizar la OMI desde el punto de vista del consumo de energía va a seguir siendo una tendencia muy importante en el futuro.

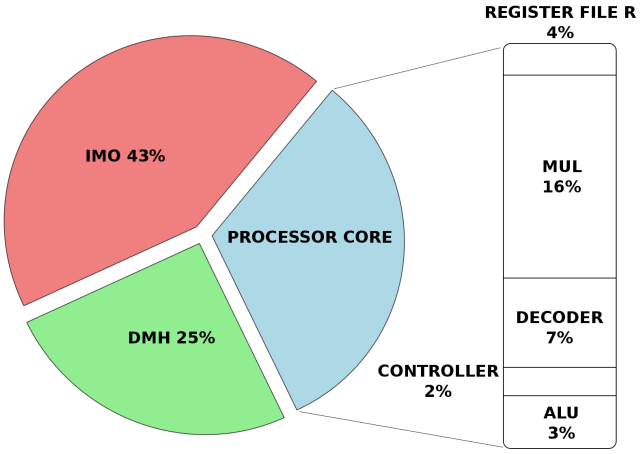


Figura A.2: Desglose de potencia de un nodo de sensor inalámbrico biomédico ejecutando un algoritmo de detección de latidos del corazón [YKH+09].

Del estudio de la literatura realizado en esta tesis doctoral (ver Capítulo 2), se puede ver que la futura investigación en la OMI se va a encontrar centrada en el desarrollo de mejoras y optimizaciones que aumenten la explotación de paralelismo dentro de la arquitectura no sólo para mejorar el rendimiento, sino también para llegar a ser más eficientes energéticamente. Con el fin de lograr este propósito, los futuros diseñadores de la OMI deberán tener en cuenta que un incremento en el paralelismo del sistema puede estar directamente relacionado con el aumento del rendimiento, pero no necesariamente con la eficiencia energética. La preocupación sobre el diseño de la OMI está apareciendo lentamente debido al hecho de que los sistemas electrónicos están empezando a ser caracterizados por poseer recursos limitados y bajos consumos de energía. Debido a este hecho, será crucial introducir mejoras en la OMI para no sólo disminuir el consumo de energía, sino también mejorar la distribución de dicho consumo a lo largo de todo el sistema. A pesar de que la arquitectura *Core 2* [INT11] posee la primera mejora arquitectural que está empezando a aplicarse, el número de implementaciones en dispositivos comerciales de las mejoras que son descritas en el Capítulo 2 será incrementado considerablemente en el futuro próximo.

A.1.3. Planteamiento del Problema

El diseñador de sistemas empotrados se enfrenta a una tarea compleja. La optimización del diseño requiere estar familiarizado con una amplia gama de tecnologías, que van desde compiladores y sistemas operativos hasta diseño lógico y empaquetado. La visión general del estado del arte (ver Capítulo 2), proporciona suficiente información para obtener la familiaridad necesaria con la amplia gama de problemas específicos a los que la comunidad investigadora ha encontrado solución. Gracias a esta visión general del estado del arte, se puede reparar en el hecho de que las técnicas existentes, que se encuentran centradas en la reducción del consumo de energía de la OMI, pueden ser clasificadas en base a las siguientes tendencias principales:

- El uso de pequeñas memorias en forma de bancos de memoria para la implementación de la caché de instrucciones L0 (es decir, la memoria de *loop buffer*).

- La mejora de la eficiencia en el particionado de la caché de instrucciones L1.

- La incorporación de mejoras en el compilador no sólo para mejorar el mapeo de la aplicación sobre la arquitectura, sino también para hacer un uso eficiente de las mejoras arquitecturales que pueden ser introducidas en la OMI.

Después de proporcionar una visión general de la situación actual de las OMIs, y dar al lector una primera idea de las características fundamentales y limitaciones en el diseño de los diversos tipos de las OMIs, se puede concluir que los siguientes problemas necesitan ser resueltos a fin de alcanzar el objetivo del diseñador de sistemas empotrados de proporcionar una OMI con un bajo consumo de energía por tarea:

- El uso de simulaciones *post-layout* para evaluar el impacto del consumo de energía en las arquitecturas de *loop buffer* introducidas en la OMI, en una evaluación experimental que posea un método estricto a fin de tener una estimación exacta de los parámetros parasitarios y de conmutación de la OMI.

- Debido a la creciente complejidad y tamaño de los sistemas empotrados, se necesita una herramienta de estimación del consumo de energía de alto nivel durante el proceso de diseño del sistema, a fin de aumentar la velocidad de simulación y el ahorro de energía de la OMI. Trabajos previos han creado métodos sofisticados de modelado para estimar con precisión el consumo de energía a nivel de sistema [BLRC05], [ANMD07], [KAA+07] y [LKY+06]. Sin embargo, estos métodos no realizan una exploración del espacio de diseño de los diferentes tipos de la OMI desde el punto de vista del consumo de energía. Esta herramienta de alto nivel debe permitir a los diseñadores de sistemas empotrados llevar a cabo un análisis de alto nivel de los compromisos de los diferentes sistemas de *loop buffer* que pueden existir en la OMI. Este análisis podrá ser utilizado no sólo para mostrar qué esquema de *loop buffer* es más adecuado para aplicaciones con cierto comportamiento, sino también para presentar el correcto proceso de diseño que un diseñador de sistemas empotrados tiene que seguir para tener una implementación eficiente de la OMI para una aplicación determinada.

- Como se puede apreciar después de comparar los diferentes tipos de esquemas de *loop buffer* que pueden ser utilizados en la OMI, los diseñadores de sistemas empotrados se enfrentan a una disyuntiva entre el consumo energético del sistema y el rendimiento que requiere la aplicación que se está ejecutando sobre este sistema. Las arquitecturas convencionales de *loop buffer* (es decir, las arquitecturas CELB y CLLB) no son lo suficientemente buenas en términos de eficiencia energética debido a la alta sobrecarga que estas arquitecturas muestran en su consumo de energía. Debido a este hecho, las arquitecturas DLB aparecen como una opción prometedora para mejorar la eficiencia energética de la OMI. Por lo tanto, existe la necesidad de proponer y analizar diferentes opciones de implementación de arquitecturas DLB para una aplicación dada.

A.2. Estudio de Técnicas para Bajo Consumo de Energía en Organizaciones de la Memoria de Instrucciones en Sistemas Empotrados

En los últimos años, la comunidad científica ha analizado e implementado mejoras en la JMD y en la red de comunicación que han dado lugar al perfeccionamiento de estos dos temas. Sin embargo, la cantidad de investigación relacionada con la optimización del consumo de energía de la OMI es escasa en comparación a las investigaciones relacionadas con los dos temas anteriores. Algunas cuestiones se han analizado en cierta medida, como las mejoras en la caché de instrucciones, en la decodificación de instrucciones, y en la planificación de instrucciones. Sin embargo, estos avances aún no han resuelto suficientemente bien el cuello de botella relacionado con el consumo de energía. En esta Sección A.2, el análisis de las técnicas utilizadas para mejorar el consumo de enegía de la OMI se encuentra dividido entre la Sección A.2.1 y la Sección A.2.2.

A.2.1. Resumen de las Optimizaciones de Hardware

Los diseñadores de sistemas empotrados personalizan la arquitectura de memoria en base a las conclusiones que se han obtenido a partir del análisis previo de las aplicaciones específicas. Varios trabajos como [HP07], [VM07], y [CRL+10] han demostrado que poseer grandes memorias se traduce en un mayor consumo de energía para el sistema. No sólo el énfasis en utilizar bajos consumos de energía es a menudo impulsado por el uso de baterías, sino también por la necesidad de utilizar un empaquetado del sistema menos costoso (por ejemplo, plástico en vez de cerámica).

J. Villarreal *et al.* [VLCV01] mostraron que el 77 % del tiempo de ejecución total de una aplicación es utilizado en bucles de menos de 32 instrucciones. Este hecho demuestra que en aplicaciones de procesamiento de señal e imágenes, una cantidad significativa del tiempo total de ejecución es usada en segmentos pequeños del programa. Si estos segmentos son almacenados en pequeñas memorias, el consumo de energía dinámica del sistema podrá reducirse significativamente. Esta observación es la base del concepto de *loop buffer*, que es un esquema arquitectural que sirve para reducir el consumo de energía dinámica en la OMI. Además, la técnica de implementar la OMI en base a bancos de memoria es identificada como un método eficaz para reducir el consumo de la energía relacionada con las corrientes de fugas [KKK02]. Aparte de la posibilidad de utilizar varios modos de funcionamiento de bajo consumo, el uso de bancos de memoria reduce la capacitancia efectiva en comparación con el uso de una única memoria monolítica.

Durante la última década, los investigadores han demostrado que el consumo de energía de la OMI no es despreciable, y que la técnica de *loop buffering* es una mejora arquitectural que permite no sólo disminuir el consumo total de energía, sino también tener una mejor distribución de este consumo en el sistema empotrado. La arquitectura CELB (*Central Loop Buffer Architecture for Single Processor Organisation*) representa la utilización más tradicional del concepto de *loop buffer*. R. S. Bajwa *et al.* [BHK+97] propusieron una mejora arquitectural que podía desconectar la lógica de búsqueda y decodificación si los bucles podían ser identificados, buscados y decodificados sólo una vez. Las instrucciones del bucle se decodificaban y se almacenaban localmente, de donde éstas eran ejecutadas. Para evitar cualquier degradación del rendimiento del sistema debido a la introducción de una arquitectura de *loop buffer*, L. H. Lee *et al.* [LMA99] implementaron una pequeña memoria de instrucciones en base a la definición, la detección y la utilización de una bifurcación de instrucciones especial. Esta mejora arquitectural no tenía ni etiquetas de direccionamiento ni bits de validación asociados a cada una de las entradas de la caché. J. Kin *et al.* [KGMS97] evaluaron la *Filter Cache*. Esta mejora se basaba en una caché pequeña de primer nivel que sacrificaba una porción de rendimiento con el fin de ahorrar energía. La memoria de programa sólo era accedida cuando se producía un fallo en la *Filter Cache*, de lo contrario, permanecía en modo de espera. Además, K. Vivekanandarajah *et al.* [VSB04] presentaron una mejora arquitectural que detectaba la oportunidad de utilizar la *Filter Cache*, y la habilitaba o deshabilitaba de forma dinámica. W. Tang *et al.* [TGN02] presentaron una DFC (*Decoder Filter Cache*) que tenía por finalidad la de reducir el uso de la lógica de búsqueda y decodificación a través del envío directo de instrucciones decodificadas al procesador. La Figura A.3 muestra la arquitectura genérica de este conjunto de OMIs. Esta arquitectura no posee particionado en la arquitectura de *loop buffer* ni tampoco en la memoria de programa, y sus conexiones internas dependen de un único componente centralizado. Debido a esto, no se puede lograr un paralelismo eficiente en la ejecución de una aplicación en este tipo de arquitecturas debido a la falta de recursos hardware.

El paralelismo es utilizado con el fin de aumentar la eficiencia del rendimiento de un sistema empotrado. Dado que los bucles son la parte más importante de un programa [VLCV01], diversas técnicas de transformaciones de bucles son aplicadas para explotar paralelismo entre bucles en arquitecturas de una solo hilo de ejecución. Sin embargo, los recursos centralizados y la comunicación global de las arquitecturas de un solo hilo de ejecución hacen que las arquitecturas CELB sean menos eficientes energéticamente cuando las técnicas de transformaciones de bucles son aplicadas sobre éstas. La arquitectura CLLB (*Clustered Loop Buffer Architecture with Shared Loop-Nest Organisation*) mitiga estos cuellos de botella. H. Zhong *et al.* [ZFMS05] presentaron una arquitectura de control distribuido para procesadores DVLIW

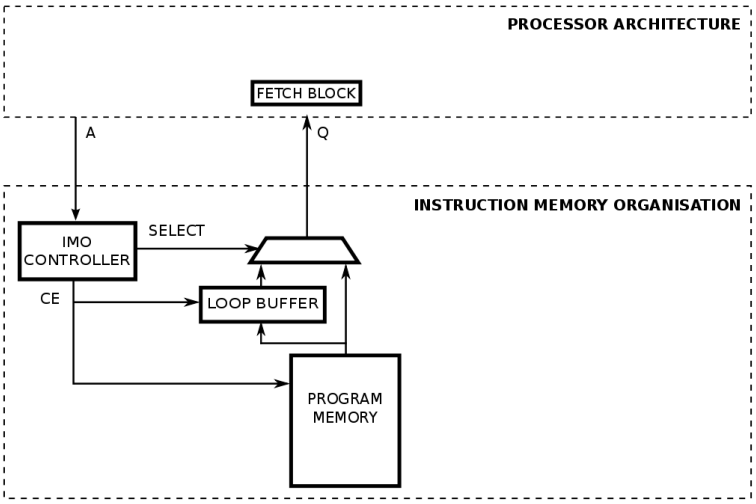


Figura A.3: Organización de la memoria de instrucciones con una arquitectura CELB.

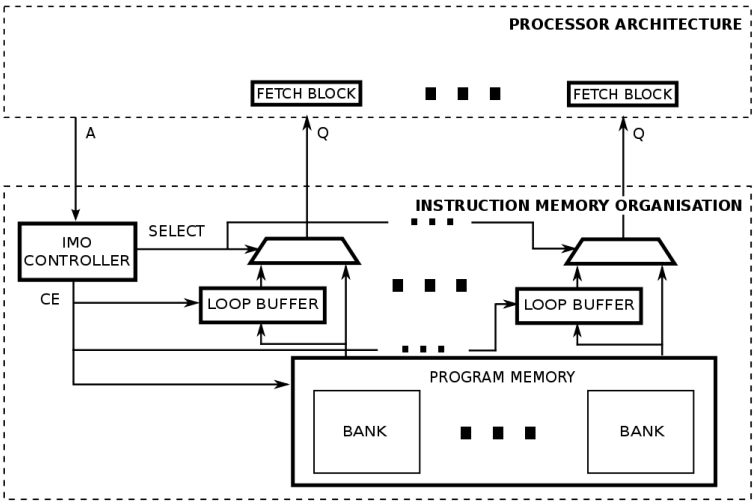


Figura A.4: Organización de la memoria de instrucciones con una arquitectura CLLB.

(*Distributed Very Long Instruction Word*), la cual resolvía el problema de escalabilidad de la lógica de control de las arquitecturas VLIW (*Very Long Instruction Word*). La mejora arquitectural se basaba en distribuir la lógica de búsqueda y decodificación de la misma manera que se distribuye en el banco de registros de un multi-*cluster* de datos. También, H. Zhong *et al.* [ZLM07] propusieron una arquitectura multi-núcleo que extendía los tradicionales sistemas basados en varios núcleos de dos maneras. En primer lugar, esta arquitectura proporcionaba una red de operación escalar de modo dual para permitir una comunicación eficiente entre núcleos sin utilizar la memoria. En segundo lugar, esta arquitectura organizaba los núcleos para su ejecución de una manera que creaba equilibrio y flexibilidad entre las diferentes latencias de las comunicaciones. D. Black-Schaffer *et al.* [BSBD+08] analizaron un conjunto de arquitecturas para la distribución eficiente de instrucciones VLIW. Una implementación de memoria caché servía de referencia para ser comparada con una variedad de OMIs, donde la evaluación incluía el coste de los accesos a la memoria y las interconexiones que eran necesarias para distribuir los bits de instrucción. La Figura A.4 muestra la arquitectura genérica de una arquitectura CLLB, cuyas conexiones internas son controladas por un solo componente. En esta arquitectura, el controlador es más complejo, debido a que controla las particiones que existen tanto en la arquitectura de *loop buffer* como en la memoria de programa. Este conjunto de arquitecturas también incluye las mejoras que provienen de la introducción de modos de funcionamiento de bajo consumo, así como de la investigación relacionada con la gestión de energía en los bancos de memorias [BMP00, FEL01, LK04].

Una explotación eficiente del paralelismo no es aún totalmente alcanzable con las arquitecturas CLLB. Con estas arquitecturas, los bucles con diferentes hilos de control tienen que ser fusionados en un solo bucle con un solo hilo de control. Sin embargo, no todos los bucles pueden ser explotados eficientemente de esta manera. En el caso de bucles incompatibles, el paralelismo no puede ser explotado de manera eficiente debido a que estos bucles requieren múltiples controladores de bucle, lo que resulta en pérdida de energía y de rendimiento. Por lo tanto, existe la necesidad de utilizar una plataforma multi-hilo, que pueda soportar la ejecución de múltiples bucles incompatibles, con una sobrecarga hardware mínima. Esto se puede lograr con la arquitectura DLB (*Distributed Loop Buffer Architecture with Incompatible Loop-Nest Organisation*). M. Jayapala *et al.* [JBB+02] propusieron una organización *cluster* de bajo consumo de energía de la memoria de instrucciones para procesadores VLIW. En la arquitectura propuesta, se utilizaba un algoritmo basado en el perfil de la aplicación para realizar una síntesis óptima de los *clusters*. P. Raghavan *et al.* [RLJ+06] presentaron una OMI distribuida de multi-hilo que soportaba la ejecución de múltiples bucles incompatibles en paralelo. En la arquitectura propuesta, cada arquitectura de *loop buffer* tenía su propio controlador local, que era responsable de la indexación y

la regulación de los accesos. J. I. Gomez *et al.* [GMV+04] presentaron una transformación de código que optimizaba el ancho de banda de la memoria, en base a la combinación de bucles con cabeceras incompatibles. Con esta técnica de transformación de código, el compilador podía entonces explotar mejor el ancho de banda disponible y aumentar el rendimiento del sistema. La Figura A.5 muestra la arquitectura genérica de esta arquitectura representativa de *loop buffer*. Como se muestra en la Figura, las conexiones internas de esta OMI son gestionadas por una lógica de controladores que se distribuye a través de la arquitectura. En esta OMI existe particionamiento tanto en la arquitectura de *loop buffer* como en la memoria de programa. El controlador de esta memoria es incluso más complejo que los controladores de las arquitecturas anteriores, debido a que éste también controla la ejecución de cada bucle en la arquitectura de *loop buffer* correspondiente, para así permitir la ejecución paralela de los bucles que poseen iteradores diferentes.

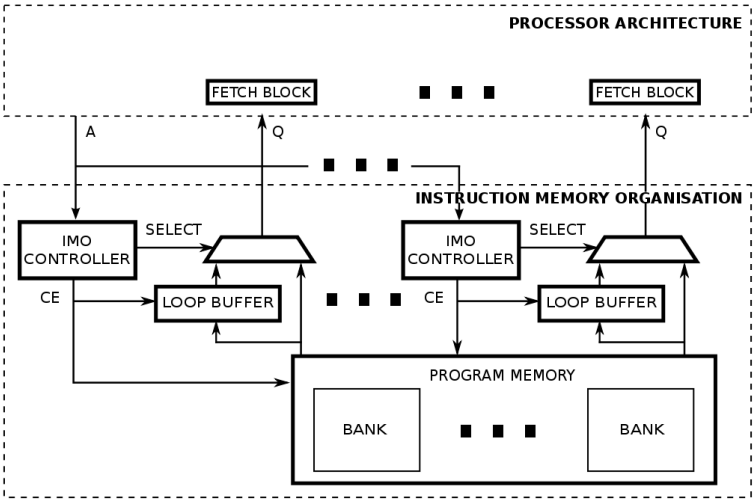


Figura A.5: Organización de la memoria de instrucciones con una arquitectura DLB.

Resumiendo, las tres arquitecturas genéricas donde toda arquitectura basada en bucles puede encajar son:

- CELB (*Central Loop Buffer Architecture for Single Processor Organisation*).
- CLLB (*Clustered Loop Buffer Architecture with Shared Loop-Nest Organisation*).
- DLB (*Distributed Loop Buffer Architecture with Incompatible Loop-Nest Organisation*).

La Sección 2.3 describe ampliamente las optimizaciones hardware de la OMI.

A.2.2. Resumen de las Optimizaciones de Software

Las características del dominio de aplicación y el diseño de la arquitectura pueden combinarse. Sin embargo, el trabajo del diseñador no ha acabado aún, porque la aplicación tiene que ser mapeada en la arquitectura. El rendimiento, el consumo de energía y la previsibilidad del comportamiento están directamente relacionados con el software que se ejecuta en el sistema empotrado. Dado que las aplicaciones empotradas hacen un uso intensivo de bucles, la reducción del consumo de energía relacionado con la ejecución de bucles es una fuente importante de beneficios.

En cuanto a la detección de perfiles para optimización de código, D. Marculescu [Mar00] abordó el problema de la optimización de energía mediante el uso de una técnica asistida por el compilador. Esta técnica se basaba en la anotación de código para seleccionar adaptadamente en tiempo de ejecución el número óptimo de instrucciones para ser buscadas o ejecutadas en paralelo. K. Inoue *et al.* [IMM02] propusieron un enfoque para detectar y eliminar innecesarias comprobaciones de etiquetas en tiempo de ejecución. Con trazas de ejecución, que estaban registradas previamente en una memoria de destino de bifurcación, era posible omitir las comprobaciones de etiquetas para todas las instrucciones contenidas en un bloque.

En relación a las transformaciones de código, T. Ishihara *et al.* [IY00] propusieron una técnica que fusionaba secuencias frecuentes de ejecución de códigos objeto en un conjunto de instrucciones individuales, lo que reducía el número de accesos a la memoria principal de forma espectacular. N. D. Liveris *et al.* [LZSG02] propusieron un flujo de diseño que aplicaba iterativamente transformaciones de código fuente para mejorar el rendimiento en la OMI. El procedimiento se basaba en un conjunto de ecuaciones analíticas que predecían el número de fallos basándose en el código de la aplicación y la estructura de la OMI.

En relación a la asignación de código, K. Pettis *et al.* [PH90] presentaron varios algoritmos de posicionamiento de código. El primer algoritmo, que fue construido sobre el enlazador, posicionaba el código en base a las llamadas a procedimientos. El segundo algoritmo, que fue construido sobre un paquete optimizador, posicionaba el código en base a los bloques básicos que existían dentro de los procedimientos. T. Vander Aa *et al.* [AJB+03] presentaron una metodología para optimizar el consumo de energía de la OMI en procesadores VLIW. Este trabajo mostró que las transformaciones de código son necesarias no sólo para aumentar la cobertura del uso de la arquitectura de *loop buffer* en la aplicación, sino también para optimizar el consumo de energía.

La Sección 2.4 describe ampliamente las optimizaciones software de la OMI.

A.3. IMOSIM: Herramienta de Exploración para la Organización de la Memoria de Instrucciones Basada en un Modelado de Energía

Como se muestra en la Sección A.1.2, el consumo de energía de la OMI de un sistema empotrado no es insignificante y necesita ser optimizado. Con el fin de reducir el consumo de energía de la OMI, los diseñadores de los sistemas empotrados modifican y/o particionan la OMI. Por un lado, *loop buffering* es un buen ejemplo de técnica efectiva basada en la modificación de la jerarquía que existe en la OMI. J. Kin *et al.* [KGMS97] mostraron que almacenando segmentos pequeños de programa en pequeñas memorias, el consumo de energía dinámico del sistema se reducía significativamente. Por otro lado, el uso de bancos de memoria es un buen ejemplo de método efectivo para el particionamiento de la OMI [KKK02]. A parte de la posibilidad de usar múltiples modos de operación de bajo consumo, el uso de bancos reduce la capacidad efectiva en comparación con el uso de una única memoria monolítica lo que lleva a mayores reducciones de energía [BMP00, FEL01, LK04].

El ITRS (*International Technology Roadmap for Semiconductors*) [ITR12] predice que el consumo de energía de los sistemas empotrados continuará su rápido crecimiento en la próxima década debido a su creciente complejidad y tamaño. A parte del consumo de energía, la fiabilidad y predictibilidad están llegando a ser también críticas, porque ambas están directamente relacionadas con el consumo de energía y su distribución sobre el sistema. Debido a esto, una herramienta de estimación de energía de alto nivel es requerida durante el proceso de diseño de un sistema empotrado para incrementar no sólo la velocidad de simulación, sino también los ahorros de energía. Trabajos previos han creado sofisticados métodos de modelado de energía que con precisión estiman el consumo de energía a nivel de sistema [BLRC05, ANMD07, KAA+07, LKY+06]. Sin embargo, estos métodos no pueden realizar una exploración del espacio de diseño de las diferentes implementaciones de la OMI desde el punto de vista del consumo de energía. En algunos casos, estos trabajos utilizan arquitecturas muy básicas de la OMI (por ejemplo, basadas en memorias caché de instrucciones L1), evitando las OMI más complejas que son potencialmente mucho más eficientes energéticamente. En otros casos, la presencia de la OMI en el sistema es completamente ignorada, porque su consumo de energía es considerado como parte del consumo de energía asignado a las operaciones que se ejecutan en la arquitectura del procesador. En esta Sección se propone una herramienta de estimación de energía de alto nivel que explora las diferentes configuraciones arquitecturales y de compilador que pueden implementar la OMI. Como puede verse en la Figura A.6, esta herramienta procesa automáticamente una aplicación determinada en base a sus características, a una arquitectura de procesador determinada y a una

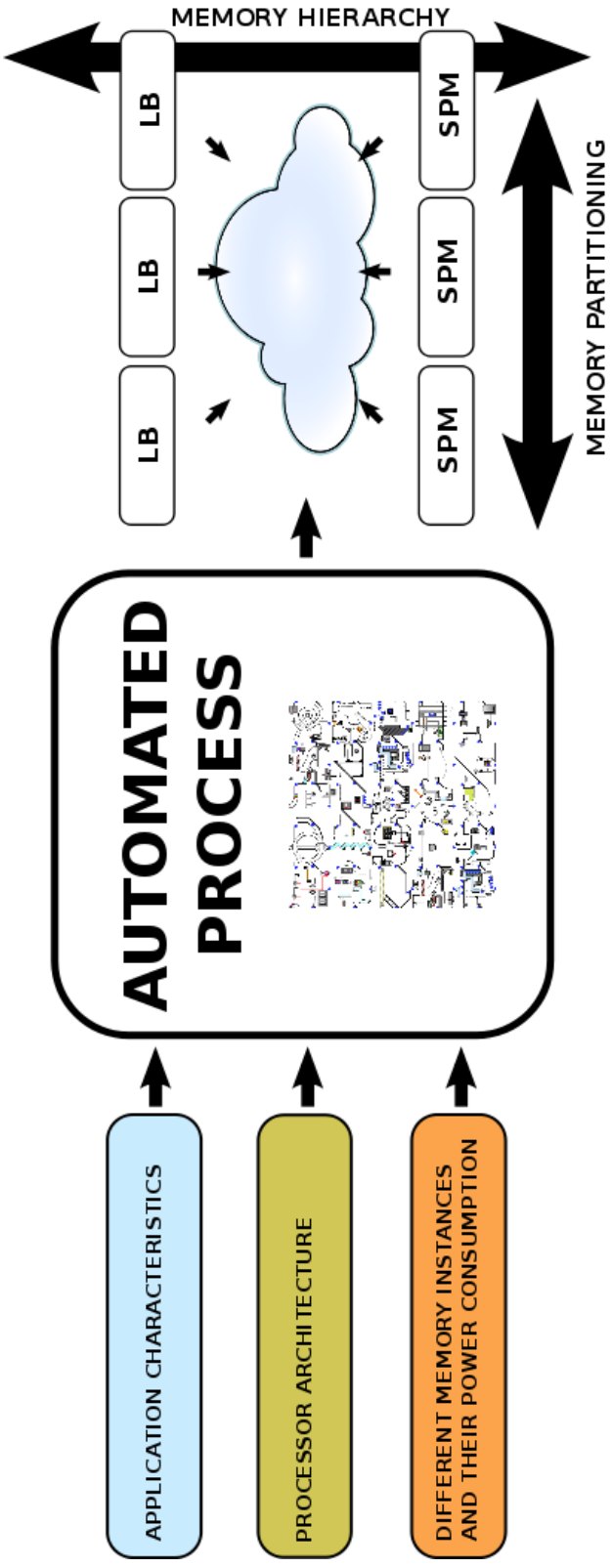


Figura A.6: Diagrama de bloques de una herramienta de exploración y estimación de energía de alto nivel.

biblioteca de consumos de energía de diferentes instancias de memoria, con el fin de ayudar a los diseñadores de sistemas empotrados a encontrar la configuración óptima de la OMI para el consumo de energía total del sistema.

A.3.1. Espacio de Diseño de la Organización de la Memoria de Instrucciones

El trabajo que se presenta en esta Sección se basa en una clasificación arquitectural que contiene tres escenarios principales donde cualquier plataforma orientada a bucles que sea eficiente desde el punto de vista del consumo de la energía puede encajar. Estos tres escenarios se encuentran representados por las arquitecturas genéricas representadas en la Figura A.3, en la Figura A.4 y en la Figura A.5 respectivamente.

La herramienta de estimación de energía que se propone en esta Sección es la única conocida por los autores que permite la simulación de la arquitectura DLB, siendo ésta la más novedosa OMI en el ámbito del consumo de energía.

A.3.2. IMOSIM

IMOSIM (*Instruction Memory Organisation SIMulator*) es una herramienta de estimación de energía de alto nivel que, para una determinada aplicación y un determinado compilador, explora diferentes arquitecturas y configuraciones que pueden componer la OMI. Esta exploración ayuda a los diseñadores de sistemas empotrados a encontrar la configuración que es óptima desde el punto de vista del consumo de energía. A fin de realizar una exploración completa del espacio de diseño de la OMI, las arquitecturas representativas descritas en la Sección A.3.1 son utilizadas para imitar todas las arquitecturas de *loop buffer* que ya están publicadas en la literatura. Como muestra la Figura A.6, IMOSIM requiere tres entradas en formato de archivo de texto para conocer la aplicación y la arquitectura del sistema empotrado: el informe histórico de los accesos a la memoria por parte de la aplicación, el informe con los requerimientos del sistema, y los modelos de energía a nivel de ciclo de cada uno de los componentes que forman las OMIs representativas. El usuario obtiene como resultado de la simulación energética no sólo los archivos de texto que contienen los perfiles de energía de los distintos componentes que forman las OMI representativas, sino también los gráficos que representan estos datos (por ejemplo, Figura A.7).

Los modelos de energía a nivel de ciclo de las diferentes instancias de memoria que forman la OMI pueden ser obtenidos de dos maneras: o bien de las hojas de características de las memorias comerciales que se desean utilizar,

o bien creando los modelos energéticos de las instancias de memoria en base a simulaciones *post-layout* de energía. En esta última opción, simulaciones RTL (*Register-Transfer Level*) tienen que ser realizadas, con algunos archivos VCD (*Value Change Dump*) de entrenamiento especificados por el usuario, para producir las precisas formas de onda de energía de la instancia de memoria que va a ser modelada.

IMOSIM procesa sus tres entradas en base a ecuaciones matemáticas que toman en cuenta el estado de la OMI en el ciclo específico de ejecución que es evaluado. Se asume que la configuración de la arquitectura de *loop buffer* es fija durante la simulación. Debido a esta limitación, sólo un conjunto limitado de divisiones puede ser elegido para un bucle específico. Una división del bucle de entrada se define formalmente en la Ecuación A.1 como un vector $S_t \epsilon \Re^n$, donde $(S_t)_i$ es el número de instrucciones que se almacenan en el banco i en el momento t, y n es el número de bancos que forman la memoria de *loop buffer*. En los modelos de energía, el tamaño de cada banco que forma la memoria de *loop buffer* se define como un vector $B_i \epsilon \Re$, donde B_i es el tamaño del banco i. Se asume que los tamaños de los bancos son continuos y van desde cero hasta un valor de tamaño máximo B_{max}. El tamaño de cada banco se selecciona con el fin de mantener la lógica de direccionamiento lo más sencilla posible. Los accesos se definen como un vector $A_t \epsilon \Re^n$, donde $(A_t)_i$ es el número de accesos al banco i en el momento t, y n es el número de bancos que forman el *loop buffer*. La Ecuación A.2 expresa la relación entre los accesos al banco $(A_t)_i$ y las posibles escisiones del bucle de entrada $(S_t)_i$.

$$0 \leq S_t \leq B_{max} \quad \forall t \tag{A.1}$$

$$A_t = S_t \iff (S_t)_i \leq B_i \quad \forall i \tag{A.2}$$

El consumo de energía de cada una de las instancias de memoria que componen la OMI se define en la Ecuación A.3 como un vector $E_i \epsilon \Re$. Por un lado, el consumo de energía dinámica $(E_{dynamic})_i$ depende de la clase de acceso $(A_t)_i$ que se lleva a cabo en B_i. Por otro lado, el consumo de energía relacionado con las corrientes de fugas $(E_{leakage})_i$ está compuesto por la energía que es consumida en los modos de funcionamiento que son activados durante el ciclo de ejecución. C_t es el vector que indica el modo de funcionamiento (*active*, *off* y *retention*) de la instancia de memoria. La salida de la simulación de IMOSIM es la evaluación de las Ecuaciones A.3, A.4 y A.5 por cada ciclo de ejecución de la aplicación en simulación.

$$E_i = (E_{dynamic})_i + (E_{leakage})_i \tag{A.3}$$

$$(E_{dynamic})_i = (E_{write})_i (A_t)_i + (E_{read})_i (A_t)_i \tag{A.4}$$

$$(E_{leakage})_i = ((E_{act})_i + (E_{off})_i + (E_{ret})_i) C_t \tag{A.5}$$

Para los sistemas empotrados de consumo de energía ultra bajo, las variaciones de temperatura que son causadas por el funcionamiento de estos sistemas son insignificantes. Además, la temperatura máxima, que estos sistemas pueden alcanzar, es menor que la temperatura en la que pueden aparecer fallos de fiabilidad. Debido al hecho de que la actual implementación de IMOSIM se encuentra centrada en los sistemas empotrados de consumo de energía ultra bajo, la dependencia térmica de las corrientes de fugas y de las variaciones PVT (*Process, Voltage, and Temperature*) no necesitan ser consideradas aún por esta versión de IMOSIM.

A.3.3. Resultados Experimentales

En el marco experimental utilizado para evaluar IMOSIM, las herramientas de *Target Compiler Technologies* [TAR12] han sido utilizadas para corroborar el correcto funcionamiento de los sistemas, así como generar el informe histórico de accesos a la memorias (ver Apéndice B). La evaluación se ha realizado usando memorias comerciales y librerías LP (*Low Power*) de 90nm de TSMC (*Taiwan Semiconductor Manufacturing Company*), y fijando la frecuencia de reloj a 100MHz. Los *benchmarks* utilizados para evaluar IMOSIM fueron seleccionados con el objetivo de mostrar la gran flexibilidad de esta herramienta. Como se puede ver en la Tabla A.1, este conjunto de *benchmarks* está formado por aplicaciones empotradas que son ejemplos excelentes, no sólo de todos los dominios de aplicación que se encuentran dominados por bucles y exhiben suficiente oportunidad de paralelismo a nivel de datos y/o de instrucción, sino también de dominios de aplicaciones de propósito más genérico que pueden ser encontrados en los diferentes tipos de procesamiento de la señal.

La salida en tiempo real de IMOSIM con el *benchmark AES* (ver Sección C.3) se presenta en la Figura A.7. Esta Figura muestra cómo el consumo de energía de la OMI cambia en cada ciclo en función de los componentes que están activos. En la región 1 de esta Figura, las instrucciones son sólo obtenidas desde la memoria de programa porque se está ejecutando una parte del programa que no forma bucle alguno. En la región 2, la OMI ha detectado que un bucle va a empezar a ejecutarse, y por lo tanto, en este caso, las instrucciones son obtenidas desde la memoria de programa para enviarlas tanto al procesador como a la arquitectura de *loop buffer*. Una vez que el bucle es almacenado en la arquitectura de *loop buffer*, la simulación se encuentra en la región 3, donde las instrucciones son obtenidas desde la arquitectura de *loop buffer* en lugar de la memoria de programa. En la última iteración del bucle, que puede ser vista en la región 4, la conexión entre la arquitectura del procesador y la memoria del programa es restaurada, de tal manera que las instrucciones siguientes sólo son obtenidas desde la memoria de programa.

Tabla A.1: *Benchmarks* utilizados en la evaluación experimental.

Benchmark [Reference]	Cycles	Issue Slots	Bits per Instruction	LB Size [Instructions]	Loop Code [%]	NOP Instructions [%]
AES [TSH+10]						
(See Section C.3)	3,347	1	16	32	77.44	0.09
BIO-IMAGING [PFH+12]						
(See Section C.4)	334,071	4	80	64	98.01	26.70
CWT NON-OPTIMISED [YKH+09]						
(See Section C.5)	274,464	1	16	32	6.61	0.48
CWT OPTIMISED [YKH+09]						
(See Section C.6)	102,827	1	20	64	6.36	2.50
DWT [DS98]						
(See Section C.7)	758,216	2	16	518	65.70	25.19
MRFA [QJ04]						
(See Section C.9)	177,170	2	32	64	19.13	17.01

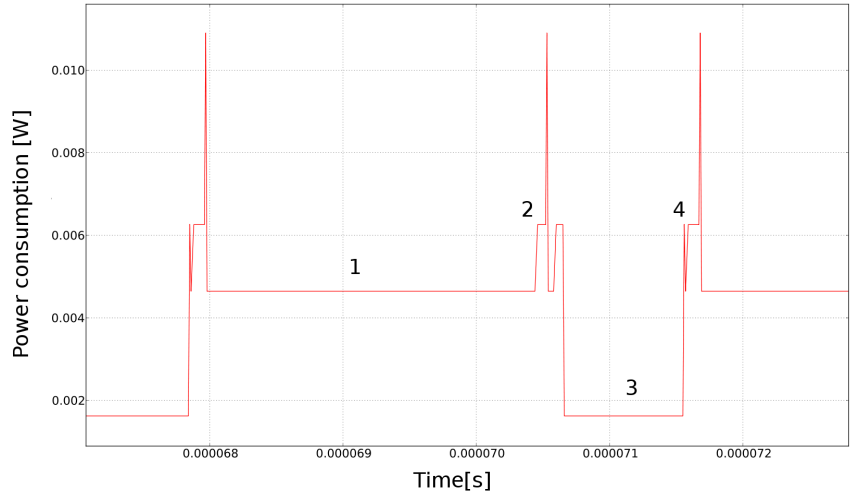


Figura A.7: Comportamiento en tiempo real de IMOSIM mostrando el consumo de potencia de una OMI basada en una arquitectura CELB.

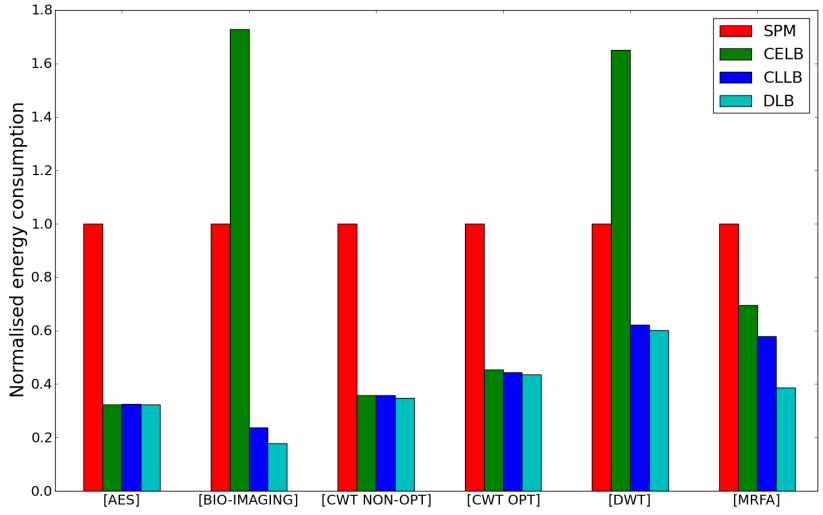


Figura A.8: Consumo de energía normalizado en diferentes OMIs ejecutando los *benchmarks* seleccionados.

La Figura A.8 contiene un diagrama de barras donde se realiza una comparación entre las diferentes OMIs. En esta Figura, podemos ver como las transformaciones de código y las configuraciones del compilador afectan al consumo de energía de la OMI. La arquitectura que sirve como referencia es una OMI basada sólo en una SPM (*Scratchpad Memory*). Como se muestra en la Figura A.8, no cualquier mejora arquitectural introducida en la OMI produce una reducción significativa del consumo de energía. La razón de las variaciones en los ahorros de energía se encuentra basada en el porcentaje de tiempo de ejecución relacionado con bucles. Si este porcentaje es bajo, los ahorros de energía son pequeños. Por un lado, la diferencia en el consumo de energía entre las arquitecturas CELB y CLLB está relacionada con las transformaciones de bucle que son aplicadas para paralizar la ejecución de la aplicación. Un paralelismo efectivo conduce a una mayor diferencia entre estas dos arquitecturas, donde la arquitectura CLLB consume menos energía. Sin embargo, un paralelismo pobre conduce a una pequeña diferencia entre ellas, e incluso, en algunos casos, la arquitectura CLLB puede consumir mayor energía que la arquitectura CELB. Por otro lado, la diferencia energética entre las arquitecturas CLLB y DLB está relacionada con el comportamiento del compilador. Si el compilador mapea la aplicación en la arquitectura de manera eficaz, la diferencia entre ahorros de energía aumentará, donde la arquitectura DLB tendrá un menor consumo de energía. Estos resultados muestran que un diseño eficiente desde el punto de vista energético tiene que tener en cuenta, no sólo las configuraciones arquitecturales utilizadas, sino también las características de la aplicación que se está ejecutando en el sistema.

Para evaluar IMOSIM se realizaron comparaciones entre esta herramienta y simulaciones *post-layout*. La Tabla A.2 muestra un error medio de 3.95 %. Este error es compensado por los ahorros de tiempo de simulación que ofrece IMOSIM, los cuales se encuentran basados en la ventaja de no realizar una síntesis de la arquitectura del sistema cada vez que ésta es modificada.

Tabla A.2: Evaluación de la precisión de IMOSIM usando una arquitectura CELB.

Benchmark [Reference]	**IMOSIM** simulation	**Post-Layout** simulation	**Error** [%]
AES [TSH+10] (See Section C.3)	5.04×10^{-05} [mJ]	4.97×10^{-05} [mJ]	1.34 %
BIO-IMAGING [PFH+12] (See Section C.4)	2.71×10^{-03} [mJ]	2.89×10^{-03} [mJ]	6.79 %
CWT NON-OPTIMISED [YKH+09] (See Section C.5)	4.44×10^{-03} [mJ]	4.35×10^{-03} [mJ]	2.02 %
CWT OPTIMISED [YKH+09] (See Section C.6)	2.08×10^{-03} [mJ]	2.15×10^{-03} [mJ]	3.56 %
DWT [DS98] (See Section C.7)	2.07×10^{-02} [mJ]	2.15×10^{-02} [mJ]	3.76 %
MRFA [QJ04] (See Section C.9)	3.18×10^{-03} [mJ]	3.38×10^{-03} [mJ]	6.23 %

A.4. Exploración del Espacio de Diseño de los Esquemas de Loop Buffer para Sistemas Empotrados

Un análisis a alto nivel de los compromisos de la OMI es crucial y necesario, ya que el espacio de diseño de las mejoras en la OMI para reducir su consumo de energía es enorme. Esta Sección presenta un análisis que propone un método para evaluar diferentes esquemas prometedores de *loop buffer*, con el fin de ayudar a los diseñadores de sistemas empotrados en la selección de la mejora que tiene que ser introducida en la OMI para una aplicación determinada. Dicha selección se realiza en base a los compromisos que existen entre el consumo de energía, el rendimiento requerido y el coste del área del sistema empotrado. Esta decisión realizada en etapas tempranas del diseño afectará dramáticamente al consumo de energía del sistema empotrado.

Tabla A.3: Consumo de potencia de bucles que son ejecutados sobre memorias de *loop buffer* de diferente tamaño.

Loop Body Size [Instruction Word]	Loop Buffer Size [Instruction Word]		
	4	16	32
4	1.02×10^{-04} [W]	1.72×10^{-04} [W]	3.73×10^{-04} [W]
16	1.05×10^{-03} [W]	1.72×10^{-04} [W]	3.73×10^{-04} [W]
32	1.05×10^{-03} [W]	1.10×10^{-03} [W]	3.73×10^{-04} [W]

A.4.1. Ejemplo Motivador

El análisis de alto nivel de los compromisos de las diferentes arquitecturas de *loop buffer* que se presentan en esta Sección difiere de anteriores trabajos como [BSBD+08] en los puntos siguientes. En primer lugar, el trabajo presentado en esta Sección sólo depende de las características de la OMI, y no se encuentra ligado a ningún procesador. En segundo lugar, no se centra en un conjunto específico de sistemas empotrados como demuestran los *benchmarks* utilizados en la Sección A.4.2. En tercer lugar, el análisis propone, no sólo el diseño arquitectural, sino también la implementación de la arquitectura. El propósito de este análisis se muestra claramente en el ejemplo realista que se ilustra en la Figura A.9. Esta Figura presenta la ejecución de un *benchmark* sintético sobre las arquitecturas representativas de *loop buffer* descritas en la Sección A.3.1, para mostrar los beneficios e inconvenientes de cada una de estas arquitecturas. Suponiendo que este *benchmark* se compone de tres bucles de 4, 16 y 32 instrucciones respectivamente, su ejecución en una arquitectura CELB (ver Figura A.3) es representada como se muestra en el caso (a) de la Figura A.9. En esta Figura, los bucles son secuencialmente asignados durante la ejecución del *benchmark*. El tamaño de la memoria de *loop buffer* es ajustado al tamaño del bucle más grande del conjunto de bucles

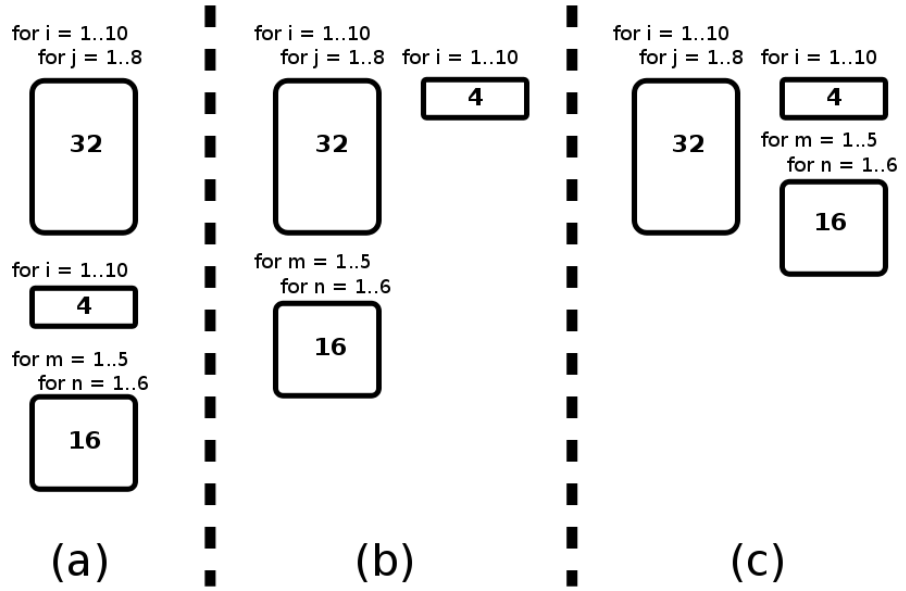


Figura A.9: Ejecución de un *benchmark* sintético en las arquitecturas representativas de *loop buffer*.

que componen este *benchmark*. Con esta estrategia, todos los bucles que constituyen el *benchmark* son almacenados sin ningún tipo de división. Si se presenta alguna división en los bucles, parte de las instrucciones deberán ser obtenidas de la memoria de programa, reduciendo el ahorro de energía de la arquitectura de *loop buffer*. Sin embargo, en esta arquitectura de *loop buffer*, no existe penalización en el rendimiento, porque se utilizan saltos en la memoria de programa para manejar esta situación. Utilizando los valores presentados en la Tabla A.3, se puede calcular utilizando la Ecuación A.6 el consumo de energía de esta arquitectura de *loop buffer* para una frecuencia específica de operación (de hecho, en este ejemplo es 100MHz). Se debe tener en cuenta que en todos los cálculos, E_{lbXLBY} es la energía que un bucle de tamaño X instrucciones consume en una arquitectura de *loop buffer* con un tamaño de instrucciones Y. Se puede ver que el cálculo de este consumo de energía E_{lbXLBY} se basa en: el consumo de energía de la arquitectura de *loop buffer*, el número total de accesos realizados en la arquitectura de *loop buffer* y el tiempo que requiere un acceso para ser realizado. Estos tres componentes están divididos por paréntesis en las Ecuaciones A.6, A.7, A.8, A.9 y A.10 para apreciar cómo cambian sus valores entre los distintos casos.

$$\begin{aligned} E_{CELB} = \; & E_{lb4LB32} + E_{lb16LB32} + E_{lb32LB32} \\ = \; & ((3.73 \times 10^{-04}) \times (4 \times 10) \times (10^{-08})) \\ + \; & ((3.73 \times 10^{-04}) \times (16 \times 30) \times (10^{-08})) \\ + \; & ((3.73 \times 10^{-04}) \times (32 \times 80) \times (10^{-08})) = 1.15 \times 10^{-08} J \end{aligned} \tag{A.6}$$

Si este *benchmark* es ejecutado en una arquitectura CLLB (ver Figura A.4), caso (b) de la Figura A.9, el primer paso es analizar las dependencias de datos que existen entre los bucles y ver qué bucles son incompatibles. En este ejemplo, como se supone que no hay dependencias de datos, sólo los bucles que tienen un tamaño de 4 y 32 instrucciones pueden ser ejecutados en paralelo. En este caso, la memoria de programa y el *loop buffer* se dividen en tamaños más pequeños e individuales para adaptarse a la necesidad de cada *cluster* de instrucción que forma la arquitectura de *loop buffer*. El bucle que tiene un tamaño de 16 instrucciones es incompatible con los otros dos, y esta arquitectura de *loop buffer* no permite la ejecución de múltiples bucles incompatibles en paralelo. Por un lado, si se utilizan los dos *loop buffers* que tienen el mismo tamaño (es decir, 32 instrucciones), el consumo de energía se estima con la Ecuación A.7. Por otro lado, si se elige adaptar el tamaño de los *loop buffers* a los bucles que se ejecutan en ellos (es decir, *loop buffers* de 4 y 32 instrucciones), el consumo de energía se estima con la Ecuación A.8. Basándose en estos resultados, se puede ver que la mejora en el ahorro de energía que proviene de la utilización de la arquitectura CLLB en lugar de la arquitectura CELB está relacionada con la mejor adaptación de los tamaños de las memorias del *loop buffers* a los tamaños de los bucles que forman la aplicación. Además, en este caso, las instrucciones NOP son obtenidas desde los bancos que forman la memoria de programa, pero no se guardan en las memorias del *loop buffer*, reduciendo las transacciones entre los componentes de la OMI. Esto introduce una penalización en el rendimiento del *benchmark* en la arquitectura CLLB.

$$\begin{aligned} E_{CLLB1} = & \quad E_{lb4LB32} + E_{lb16LB32} + E_{lb32LB32} \\ = & \quad ((3.73 \times 10^{-04}) \times (4 \times 10) \times (10^{-08})) \\ + & \quad ((3.73 \times 10^{-04}) \times (16 \times 30) \times (10^{-08})) \\ + & \quad ((3.73 \times 10^{-04}) \times (32 \times 80) \times (10^{-08})) = 1.15 \times 10^{-08} J \end{aligned} \tag{A.7}$$

$$\begin{aligned} E_{CLLB2} = & \quad E_{lb4LB4} + E_{lb16LB32} + E_{lb32LB32} \\ = & \quad ((1.02 \times 10^{-04}) \times (4 \times 10) \times (10^{-08})) \\ + & \quad ((3.73 \times 10^{-04}) \times (16 \times 30) \times (10^{-08})) \\ + & \quad ((3.73 \times 10^{-04}) \times (32 \times 80) \times (10^{-08})) = 1.14 \times 10^{-08} J \end{aligned} \tag{A.8}$$

Por último, si el *benchmark* es ejecutado en una arquitectura DLB (ver Figura A.5), el primer paso es, de nuevo, analizar las dependencias de datos entre bucles, pero en este caso, no hay necesidad de comprobar si los bucles son incompatibles, ya que este tipo de arquitectura de *loop buffer* permite la ejecución de múltiples bucles incompatibles en paralelo. Este escenario es mostrado en el caso (c) de la Figura A.9. También, en este último caso, la memoria de programa y el *loop buffer* son divididos en tamaños más pequeños e individuales para adaptarse a la necesidad de cada *cluster* de instrucción

que forma la arquitectura de *loop buffer*. Por un lado, si se utilizan dos *loop buffers* con el mismo tamaño (es decir, 32 instrucciones), el consumo de energía se estima con la Ecuación A.9. Por otro lado, si se elige adaptar el tamaño de los *loop buffers* a los bucles que se ejecutan en ellos (es decir, *loop buffers* de 16 y 32 instrucciones), el consumo de energía se estima con la Ecuación A.10. Basándose en estos resultados, es posible concluir que cualquier mejora del paralelismo en el sistema no sólo aporta mejoras en el rendimiento, sino también mejoras en el consumo de energía del sistema. El aumento del ILP (*Instruction-Level Parallelism*) facilita la adaptación de las dimensiones del *loop buffer* a los tamaños de los bucles que forman la aplicación, ya que da más libertad para combinar la ejecución de los bucles que forman la aplicación. En esta última arquitectura de *loop buffer*, la lógica de control es más compleja que la de los controladores de las arquitecturas anteriores. Sin embargo, esta lógica de control reduce considerablemente el tiempo de ejecución y el consumo de energía total de la aplicación, ya que permite la ejecución en paralelo de bucles con iteradores diferentes.

$$\begin{aligned} E_{DLB1} = & \quad E_{lb4LB32} + E_{lb16LB32} + E_{lb32LB32} \\ = & \quad ((3.73 \times 10^{-04}) \times (4 \times 10) \times (10^{-08})) \\ + & \quad ((3.73 \times 10^{-04}) \times (16 \times 30) \times (10^{-08})) \\ + & \quad ((3.73 \times 10^{-04}) \times (32 \times 80) \times (10^{-08})) = 1.15 \times 10^{-08} J \end{aligned} \tag{A.9}$$

$$\begin{aligned} E_{DLB2} = & \quad E_{lb4LB16} + E_{lb16LB16} + E_{lb32LB32} \\ = & \quad ((1.72 \times 10^{-04}) \times (4 \times 10) \times (10^{-08})) \\ + & \quad ((1.72 \times 10^{-04}) \times (16 \times 30) \times (10^{-08})) \\ + & \quad ((3.73 \times 10^{-04}) \times (32 \times 80) \times (10^{-08})) = 1.04 \times 10^{-08} J \end{aligned} \tag{A.10}$$

Como se puede ver después de comparar las arquitecturas representativas de *loop buffer*, los diseñadores de sistemas empotrados pueden enfrentarse al compromiso existente entre el consumo total de energía del sistema empotrado y el rendimiento requerido por la aplicación que se ejecuta en éste. Como se muestra en este ejemplo, las arquitecturas convencionales de *loop buffer* (es decir, las arquitecturas CELB y CLLB) no son lo suficientemente buenas en términos de eficiencia energética debido a los costes que estas arquitecturas muestran. Debido a este hecho, las arquitecturas DLB aparecen como una opción prometedora a fin de mejorar la eficiencia energética de la OMI. La Sección A.4.2 no sólo presenta el análisis sistemático de este compromiso, sino también demuestra que la sobrecarga de área y la tecnología seleccionada para la implementación de la arquitectura de *loop buffer* tienen que ser tenidas en cuenta en el presente compromiso.

A.4.2. Resultados Experimentales

El marco experimental creado para el caso de estudio y los *benchmarks* que son presentados en esta Sección consta de una JMD, una OMI, una arquitectura de procesador y un interfaz de E/S. Las herramientas de *Target Compiler Technologies* [TAR12] se han utilizado para corroborar el correcto funcionamiento de los sistemas, así como generar el informe histórico de accesos a la memorias (ver Apéndice B). En este marco experimental, la arquitectura de *loop buffer* consta de una memoria y un controlador. La implementación de la memoria de la arquitectura de *loop buffer* se basa en un conjunto de bancos, en el que cada banco puede ser configurado para ajustarse al tamaño deseado. El controlador de la arquitectura de *loop buffer* es el componente que controla el estado de la memoria de la arquitectura de *loop buffer* dentro de la OMI. Las simulaciones energéticas, que se muestran en esta Sección, se han llevado a cabo utilizando la herramienta de estimación de energía de alto nivel descrita en la Sección A.3. La evaluación se ha realizado usando memorias comerciales y librerías LP (*Low Power*) de 90nm de TSMC (*Taiwan Semiconductor Manufacturing Company*), y fijando la frecuencia de reloj a 100MHz. La Tabla A.1 muestra los *benchmarks* utilizados en esta Sección. Este conjunto de *benchmarks* está formado por aplicaciones empotradas que son ejemplos excelentes, no sólo de todos los dominios de aplicación que se encuentran dominados por bucles y exhiben suficiente oportunidad de paralelismo a nivel de datos y/o de instrucción, sino también de dominios de aplicaciones de propósito más genérico que pueden ser encontrados en los diferentes tipos de procesamiento de la señal. Esta selección de *benchmarks* se realizó con el objetivo de hacer que el análisis que se presenta en esta Sección sea lo suficientemente genérico, a fin de ser aplicable a todos los dominios de aplicación de los sistemas empotrados que se encuentran dominados por bucles.

A.4.2.1. Variación Energética Influenciada por el Manejo de Condiciones

Cuando la OMI no tiene arquitectura de *loop buffer* y sólo se compone de una única SPM, cada instrucción tiene que ser obtenida de esta memoria de programa, incluso las instrucciones NOP. En este caso, todos los *cluster* de instrucción deben tener la misma estructura de condición. Esta estructura de condición se basa normalmente en dos opciones: predicación o salto en la memoria de programa. Por un lado, si se basa en predicación, la rama de acierto y la de fallo se ejecutan en paralelo. Por otro lado, si se basa en salto en la memoria de programa, tienen que hacerse saltos en la memoria de programa hacia las direcciones adecuadas. Por lo tanto, predicación aumenta el consumo de energía y el tiempo de ejecución del sistema empotrado en comparación

con saltos en el memoria de programa. La Figura A.10 muestra el consumo de energía de estas arquitecturas como referencia para la comparación con las arquitecturas CELB y CLLB.

Si en la OMI se utiliza una arquitectura CELB, el tamaño total de almacenamiento de la OMI se incrementa hasta un 25 % en comparación con la OMI que se basa en una SPM, debido al aumento de los componentes y conexiones que forman la OMI por la introducción de la arquitectura de *loop buffer*. Sin embargo, como se puede ver en la Figura A.10, el aumento del número de accesos a los componentes del interior de la OMI no aumenta el consumo de energía total de la OMI. En efecto, este consumo de energía es menor (hasta un 67 %), porque la mayor parte de los accesos que se hacen en la OMI se realizan en la arquitectura de *loop buffer*. En base a la Tabla A.1, es posible ver que la reducción de energía no sólo depende del porcentaje de tiempo total de ejecución relacionada con bucles, sino también del tamaño de *loop buffer* que es usado. Es evidente que debe ser tomado en cuenta el número de iteraciones de los bucles para evaluar si tiene sentido introducir una arquitectura de *loop buffer*. Los *benchmarks BIO-IMAGING* y *DWT* son excepciones a esta regla, como se explica en la Sección A.4.2.2. La arquitectura CELB tiene la misma estructura de condición que la OMI basada en una única SPM. Sin embargo, en este caso, saltos en la arquitectura de *loop buffer* pueden ser soportados. Si no son soportados, la OMI aumenta su consumo de energía porque la arquitectura de *loop buffer* no es utilizada debido a los saltos y los accesos que tienen que ser realizados en la memoria de programa. Por lo tanto, predicación y saltos en la memoria del programa son las opciones que aumentan fuertemente el consumo de energía del sistema empotrado en comparación con saltos en la arquitectura de *loop buffer*.

Si la OMI incluye una arquitectura CLLB, la memoria de programa y la arquitectura de *loop buffer* se dividen en bancos más pequeños e individuales para ajustarse a las necesidades de cada *cluster* de instrucción que forma la arquitectura. En este caso, el tamaño de almacenamiento total de la OMI no se modifica en comparación con el caso de la arquitectura CELB, porque se almacenan instrucciones iguales dentro de un bucle múltiples veces. Debido al hecho de que en esta OMI las instrucciones NOP son obtenidas desde los bancos que forman la memoria de programa, pero no son guardadas en la arquitectura de *loop buffer*, el número de instrucciones que tienen que ser obtenidas y guardadas dentro de la OMI disminuye. Como se muestra en la Figura A.10, la arquitectura CLLB disminuye el consumo total de energía (42 % en media) de la OMI en comparación con la arquitectura CELB, y esta disminución depende directamente del número de instrucciones NOP que están contenidas en el código de los bucles. Además, en este caso, por defecto se soportan saltos en la arquitectura de *loop buffer*. Sin embargo, en

este caso, hay más opciones en cómo se manejan las condiciones. Si existen diferentes condiciones a través de todos los *loop buffers*, es necesario hacer que estas condiciones sean compatibles en todos los *loop buffers*. Con el fin de resolver este problema, más instrucciones NOP tienen que estar presentes en la arquitectura de *loop buffer*, pero estas no obtenidas por el procesador. Se puede apreciar el impacto de la manipulación de condiciones en los *benchmarks BIO-IMAGING* y *DWT*, donde la manipulación de condiciones utilizada por la arquitectura CLLB mejora considerablemente el consumo de energía de la OMI en comparación con la manipulación de condiciones utilizada por la arquitectura CELB.

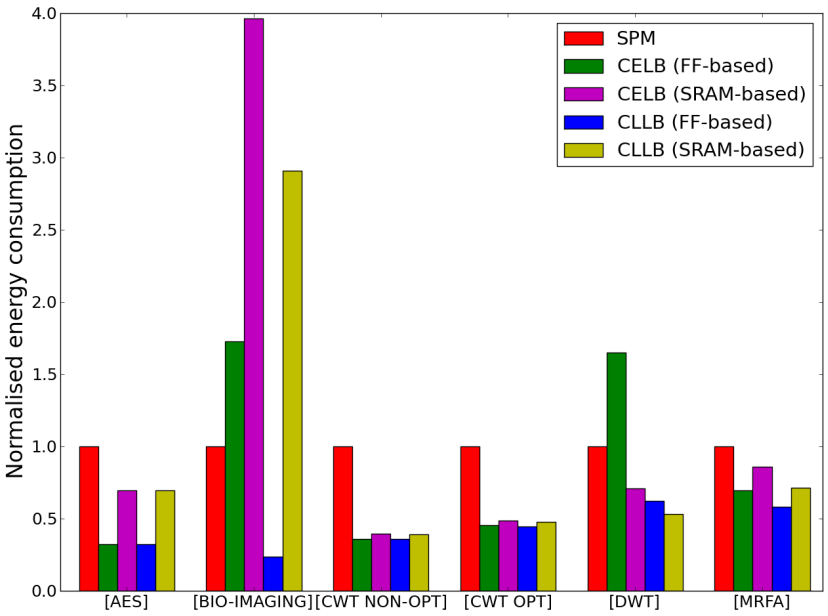


Figura A.10: Consumo de energía normalizado en diferentes OMIs ejecutándose los *benchmarks* seleccionados.

A.4.2.2. Variación Energética Influenciada por la Tecnología

En la Sección A.4.2.1, se puede ver que cuando la OMI utiliza una arquitectura CELB, en el caso de los *benchmarks BIO-IMAGING* y *DWT*, el aumento del número de accesos a los componentes internos de la OMI aumenta el consumo de energía total de la OMI. Para analizar este hecho, la Figura A.11 presenta una implementación basada en FFs y una implementación basada en SRAM de la arquitectura CELB. Si se compara el consumo de energía de estas dos implementaciones a lo largo de todos los *benchmarks* utilizados en esta

Sección, es posible ver que la implementación basada en FFs es más eficiente energéticamente que la implementación basada en SRAM en aplicaciones con requisitos de almacenamiento limitado. Sin embargo, debido a los tamaños más grandes de los *loop buffers* utilizados en los *benchmarks BIO-IMAGING* y *DWT*, las implementaciones basadas en SRAM son las mejores opciones para estos casos. Por lo tanto, la tecnología de almacenamiento juega un papel importante. Y cada vez que sea factible, el tamaño del código que va a ser almacenado debe ser reducido mediante la aplicación adecuada de transformaciones de código y optimizaciones del compilador.

La Figura A.11, basada en curvas de Pareto que relacionan la energía y el área, muestra para dos *benchmarks* de la Tabla A.1 el incremento del área de la memoria que se requiere para lograr cierto ahorro de energía. Como se muestra en esta Figura, la penalización en área, que el diseñador de sistemas empotrados tiene que asumir para lograr un mayor ahorro de energía es relativamente pequeño (10 % en promedio). Pero ésto muestra que un compromiso interesante tiene que ser decidido en base al contexto completo del diseño. Las características de la aplicación (es decir, el número de instrucciones NOP en bucles) tienen que ser analizadas para tomar la decisión en este compromiso. Si la aplicación tiene en los bucles un alto porcentaje de instrucciones NOP, la arquitectura CLLB tiene que ser seleccionada. Si este no es el caso, la arquitectura CELB tiene que ser seleccionada.

A.4.2.3. Variación Energética Influenciada por Transformaciones de Código y Compilador

La Figura A.12 contiene un diagrama de barras en el que se compara la arquitectura DLB con las arquitecturas CELB y CLLB. En esta Figura, se puede ver cómo las transformaciones de código y las configuraciones del compilador afectan al consumo de energía de la OMI. La arquitectura de referencia es de nuevo una OMI basada sólo en una SPM. Como se muestra en la Figura A.12, por una parte, la diferencia en el consumo de energía entre las arquitecturas CELB y CLLB está relacionada con las transformaciones de bucles que se emplean en la aplicación para paralelizar su ejecución. Un paralelismo efectivo conduce a una mayor diferencia entre estas dos arquitecturas, donde la arquitectura CLLB tiene menos consumo de energía. Sin embargo, un paralelismo pobre dará lugar a pequeñas diferencias entre ellas, e incluso, en algunos casos, la arquitectura CLLB podría tener un mayor consumo de energía que la arquitectura CELB. Por otra parte, la diferencia en los consumos de energía entre las arquitecturas CLLB y DLB está relacionada con el comportamiento del compilador. Si el compilador mapea eficazmente la aplicación en la arquitectura, la diferencia entre los ahorros de energía se incrementará, donde la arquitectura DLB tendrá menor consumo de energía.

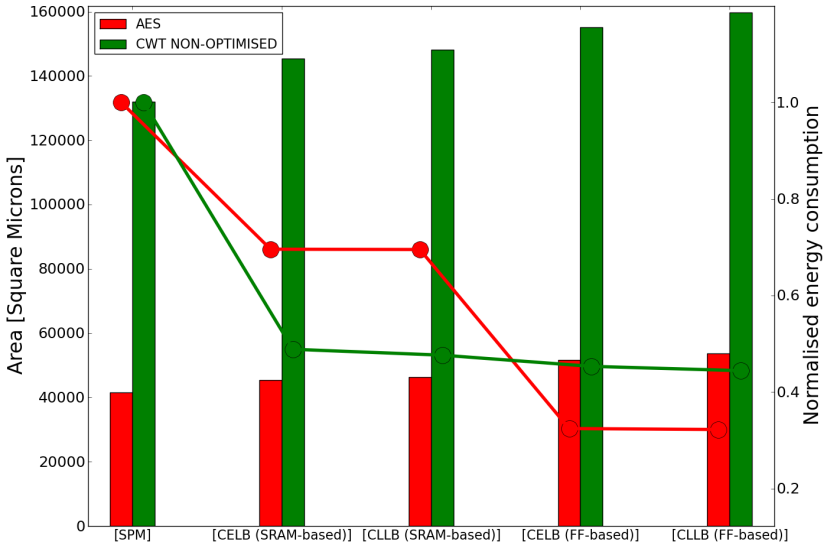


Figura A.11: Consumo de energía normalizado (representado por lineas) vs. ocupación (representada por barras) en diferentes OMIs.

Estos resultados muestran que un diseño eficiente de sistemas empotrados debe tener en cuenta no sólo las configuraciones arquitecturales que se utilizan en el sistema empotrado, sino también las características de la aplicación que se ejecuta sobre él.

A.4.2.4. Discusión y Resumen de los Compromisos Pareto-Óptimos para los Diseñadores de Sistemas Empotrados

De los resultados obtenidos a lo largo de esta Sección, se puede concluir que cada arquitectura de *loop buffer* es claramente óptima para un determinado patrón de código de la aplicación y un conjunto específico de requisitos del sistema empotrado. Debido a este hecho, un análisis de energía de alto nivel tiene que ser realizado en los primeros pasos del proceso de diseño por los diseñadores de sistemas empotrados, para no sólo aumentar el ahorro de energía, sino también tener una mejor distribución del presupuesto de energía a lo largo de todo el sistema empotrado.

El autor de esta tesis doctoral propone que el análisis energético de alto nivel tiene que ser realizado en base a las siguientes directrices:

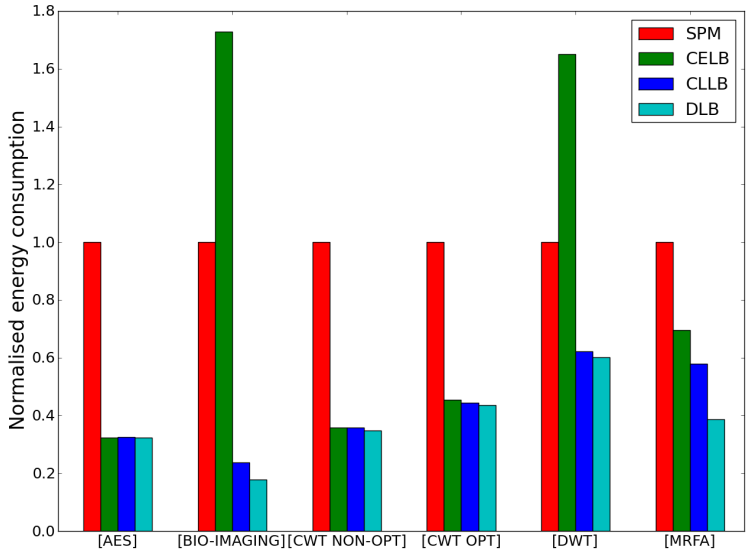


Figura A.12: Consumo de energía normalizado de las arquitecturas CELB, CLLB y DLB que son usadas con los *benchmarks* seleccionados.

- **Uso de una arquitectura de *loop buffer* en la OMI.** Para introducir una arquitectura de *loop buffer* no sólo el porcentaje de tiempo total de la ejecución que está relacionado con bucles ha de tenerse en cuenta, sino también el tamaño y el ancho de las memorias de *loop buffer* que son introducidas en la OMI (ver Sección A.4.2.2). Los ahorros de energía que se obtienen a partir de la introducción de la arquitectura de *loop buffer* tienen que compensar el consumo de energía relacionado con el aumento de los componentes y de las conexiones que forman la OMI.
- **Uso de arquitecturas CELB.** El uso de arquitecturas CELB en la OMI está relacionado con el paralelismo que puede ser explotado en el sistema. Si no hay paralelismo o existe un paralelismo pobre, las arquitecturas CELB son la mejor opción. De hecho, la diferencia que puede existir entre los consumos de energía de la arquitectura CELB y el resto de arquitecturas de *loop buffer* está directamente relacionada con las transformaciones de bucle que son empleadas en la aplicación a fin de obtener un paralelismo efectivo en su ejecución.
- **Uso de arquitecturas CLLB.** El uso de arquitecturas CLLB en la OMI no depende sólo del paralelismo que puede ser explotado en la aplicación, sino también del número de instrucciones NOP que están contenidas en los bucles. Si la aplicación presenta un elevado número de instrucciones NOP y se puede lograr una alta eficiencia en la explotación de paralelismo, esta arquitectura de *loop buffer* es la mejor opción.

- **Uso de arquitecturas DLB.** El uso de arquitecturas DLB en la OMI no depende sólo del paralelismo y del número de instrucciones NOP que están contenidas en los bucles, sino también del comportamiento del compilador. Si el compilador puede mapear la aplicación en la arquitectura de manera efectiva, la arquitectura DLB será la arquitectura de *loop buffer* que tenga menor consumo de energía.

En el caso de que el diseño de un sistema empotrado tenga en cuenta no sólo la reducción del consumo de energía de la OMI, sino también la ocupación de área y la penalización de rendimiento que pueden aparecer cuando una arquitectura de *loop buffer* es introducida, las siguientes ventajas y desventajas tienen que ser consideradas:

- **Ocupación de área vs. consumo de energía.** Como se puede ver en la Figura A.13, cualquier mejora arquitectural que se introduce en la OMI produce un aumento en la ocupación de área. De esta Figura, también se puede ver que cuando la complejidad de la arquitectura de *loop buffer* aumenta en términos del número de componentes e interconexiones, la ocupación de área del sistema empotrado también aumenta. Sin embargo, esta Figura muestra claramente que las arquitecturas más complejas son potencialmente mucho más eficientes.

- **Penalización de rendimiento vs. consumo de energía.** La Figura A.14 muestra que no existe penalización en el tiempo de ejecución de una aplicación cuando una arquitectura de *loop buffer* es introducida en la OMI. De hecho, como se muestra en la Figura A.14, el aumento de la complejidad de la arquitectura de *loop buffer* reduce el tiempo de ejecución. Esto está directamente relacionado con la alta eficiencia del paralelismo que pueden lograr las arquitecturas de *loop buffer* complejas. Además, en esta disyuntiva, la estructura de condición que se utiliza en el procesador tiene que también tenerse en cuenta. Como se muestra en la Sección A.4.2.1, predicación y saltos en la memoria de programa son opciones que aumentan fuertemente el consumo de energía y tiempo de ejecución del sistema empotrado en comparación con saltos en la arquitectura de *loop buffer*.

En base a las directrices anteriores, la OMI óptima energéticamente para una aplicación específica y un sistema empotrado determinado tiene que ser seleccionada entre las implementaciones complementarias de las arquitecturas de *loop buffer* que cubren las distintas particiones del espacio de diseño de la OMI. Con el fin de encontrar esta OMI óptima, tienen que decidirse interesantes disyuntivas basándose en el contexto global del diseño, es decir, en base a las características del código de la aplicación y a la arquitectura de procesador que forman el sistema empotrado.

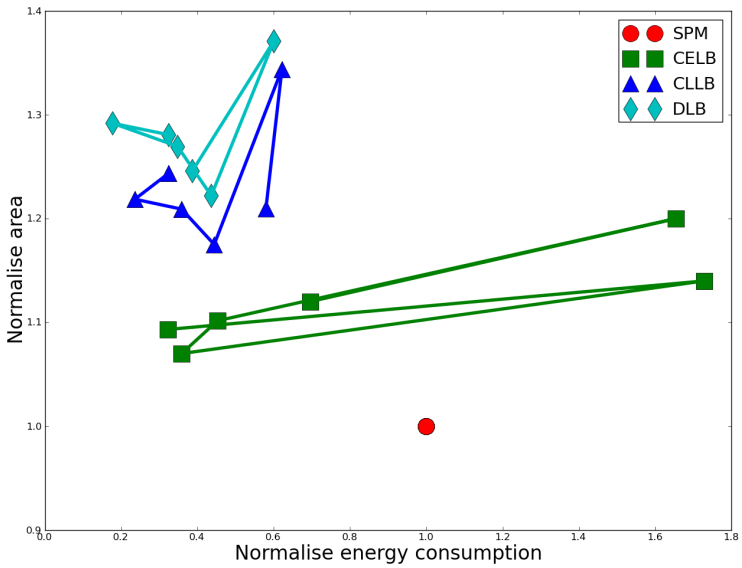


Figura A.13: Consumo de energía normalizado vs. ocupación de área de los *benchmarks* seleccionados en las diferentes OMIs representativas.

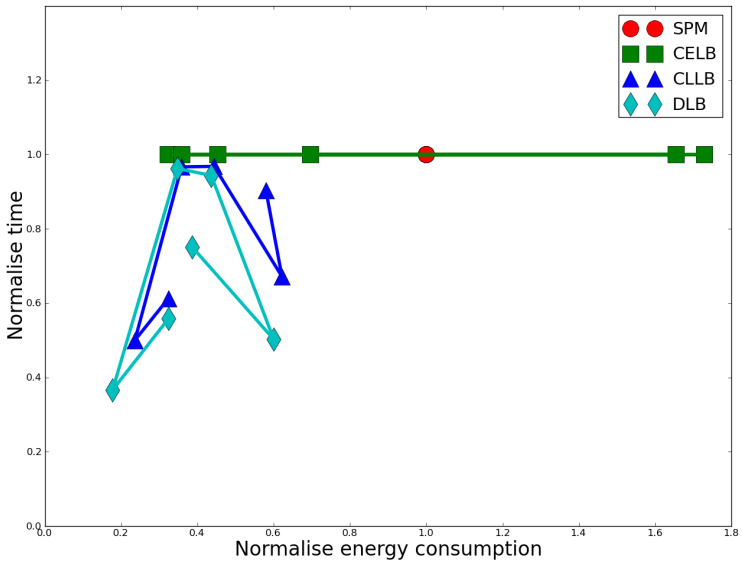


Figura A.14: Consumo de energía normalizado vs. penalización en rendimiento de los benchmarks seleccionados en las diferentes OMIs representativas.

A.5. Caso de Estudio del Impacto de Potencia de los Esquemas de Loop Buffer para Nodos de Sensores Inalámbricos Biomédicos

Los sistemas empotrados constituyen el dominio digital de los nodos de una WSN (*Wireless Sensor Network*). Éstos son utilizados ampliamente en varios tipos de ámbitos que van desde la monitorización industrial hasta aplicaciones biomédicas. En particular, para el ámbito biomédico, la información que se procesa es confidencial y requiere autenticación para transmitirse. Debido a este hecho, no es raro que dos aplicaciones como un algoritmo HBD (*Heartbeat Detection*) y un algoritmo criptográfico como AES (*Advanced Encryption Standard*) se puedan encontrar en los nodos de una WSN biomédica. Estas dos aplicaciones empotradas de la vida real se utilizan en esta Sección como casos de estudios, a fin de evaluar los ahorros de energía logrados por el uso de OMIs basadas en el concepto de *loop buffer*. Las arquitecturas de *loop buffer* que se analizan en esta Sección son las arquitecturas CELB (*Central Loop Buffer Architecture for Single Processor Organisation*) y BCLB (*Banked Central Loop Buffer Architecture*).

Las arquitecturas CELB y CLLB se pueden implementar usando bancos de memoria o sin ellos. La gestión del consumo de potencia usando bancos de memorias se ha investigado desde diferentes ángulos, incluyendo hardware, sistema operativo y compilador. Con patrones de acceso a la memoria, L. Benini *et al.* [BMP00] propusieron un algoritmo para dividir la SRAM (*Static Random Access Memory*) interna en múltiples bancos que podían ser accedidos de forma independiente. X. Fan *et al.* [FEL01] presentaron políticas para el controlador de la memoria de OMIs con modos de funcionamiento de bajo consumo de energía. C. Lyuh *et al.* [LK04] utilizaron un compilador que determinaba los modos de funcionamiento de los bancos de memoria después de planificar las operaciones de la memoria. Como se puede ver de los enfoques anteriores, el inconveniente de la utilización de múltiples *loop buffers* es por lo general el aumento de la lógica que controla los bancos, sin embargo tiene el beneficio de reducir el consumo de energía relacionado con las corrientes de fuga. Este hecho conduce al aumento de las capacidades de interconexión, así como a la reducción de los posibles ahorros de energía dinámica que están relacionados con el acceso a memorias pequeñas. La mayoría de los enfoques que están relacionados con cachés asumen que la sintonización automatizada se realiza estáticamente, lo que significa que la puesta a punto se realiza durante el diseño de la aplicación. A. Ghosh *et al.* [GG04] presentaron un heurístico que, a través de un modelo analítico, determinaba las configuraciones de la memoria caché basándose en las restricciones de la aplicación. Podían realizarse otros enfoques de ajuste de caché de forma dinámica mientras una aplicación era ejecutada [GRVD09].

A.5.1. Marco Experimental

En el marco experimental utilizado para evaluar los esquemas de *loop buffer* descritos en esta Sección, las herramientas de *Target Compiler Technologies* [TAR12] (ver Apéndice B) han sido utilizadas para diseñar los procesadores descritos en Sección A.5.1.1 y Sección A.5.1.2.

A.5.1.1. Procesador Optimizado para el Algoritmo HBD

La arquitectura de procesador optimizada para el algoritmo HBD se basa en la arquitectura de procesador que se presenta en la Sección B.3. Esta Sección presenta las modificaciones y las optimizaciones que se realizan sobre la arquitectura de procesador de propósito general para construir la arquitectura del procesador optimizada, la cuál es mostrada en la Figura C.26. Básicamente, se añaden un nuevo generador de direcciones (AG2) y una segunda ALU (ALU 2), además de algunos puertos. Aparte de eso, el contador de programa es modificado para manejar palabras de instrucción que utilizan valores inmediatos de 32 bits. Con el fin de manejar señales ECG muestreadas a 1KHz, las memorias que se requieren por esta arquitectura de procesador son una DM con una capacidad de 8K palabras de 32 bits cada una, y una CM (*Constant Memory*) con un capacidad de 8K palabras de 32 bits. Además, la memoria de programa que se requiere por esta arquitectura de procesador es una memoria con una capacidad de 1K palabras de 20 bits. Esta arquitectura de procesador optimizada se encuentra basada en el trabajo que se presenta en la referencia [YKH+09].

A.5.1.2. Procesador Optimizado para el Algoritmo AES

La arquitectura de procesador optimizada para el algoritmo AES también se basa en la arquitectura de procesador que se presenta en la Sección B.3. Esta Sección presenta las modificaciones y las optimizaciones que se realizan sobre la arquitectura de procesador de propósito general para construir la arquitectura del procesador optimizada, la cuál es mostrada en la Figura C.6. Básicamente, se añade una ruta de datos adicional de 128 bits, la cuál incluye una VM (*Vector Memory*), un V (*Vector Register File*), y una unidad vectorial (Vector Unit functional). Por un lado, para manejar una señal de entrada de $1,460$ bytes, la memoria de datos que se requiere por esta arquitectura de procesador es una memoria con una capacidad de 1K palabras de 16 bits cada una, y la VM requerida es una memoria con una capacidad de 64 palabras de 128 bits. Por otro lado, la memoria de programa requerida es una memoria con una capacidad de 1K palabras de 16 bits cada una. Esta arquitectura de procesador optimizada se encuentra basada en el trabajo que se presenta en la referencia [TSH+10].

A.5.1.3. Plataforma Experimental

La plataforma experimental se genera automáticamente para cualquiera de las arquitecturas de procesador descritas en Sección A.5.1.1 y Sección A.5.1.2. La plataforma experimental esta compuesta por una JMD, una OMI, una interfaz de E/S, y una arquitectura de procesador que es utilizada como núcleo de la plataforma del sistema empotrado. Por un lado, la memoria de programa y la memoria de datos son memorias SRAM diseñadas por *Virage Logic Corporation* tools [VIR12]. Por otro lado, la interfaz de E/S que permite recibir y enviar datos en tiempo real está directamente conectada con el banco de registros de la arquitectura de procesador, a través de las arquitecturas FIFO (*First In, First Out*) que implementan la interfaz de E/S de la arquitectura de procesador.

En la Figura A.15 se representa la interfaz entre una arquitectura de procesador y una OMI. Las interconexiones del procesador, de la memoria de programa y de la memoria y controlador del *loop buffer* están incluidas en esta Figura. Cada uno de estos componentes que forman la OMI es explicado en los párrafos siguientes. En esta plataforma experimental, la arquitectura de *loop buffer* puede ser configurada para ser utilizada como una arquitectura CELB o BCLB. La Figura A.15 muestra como la OMI puede estar formada o bien por diferentes instancias de memoria, o bien por una única instancia de memoria gracias a los multiplexores existentes en la arquitectura de *loop buffer*. La lógica circuital que decide qué parte de la arquitectura de *loop buffer* es activada será más compleja en el caso de una arquitectura CELB que en la arquitectura BCLB. Por simplicidad, la arquitectura CELB es utilizada en los párrafos siguientes para explicar la operación del concepto de *loop buffer*.

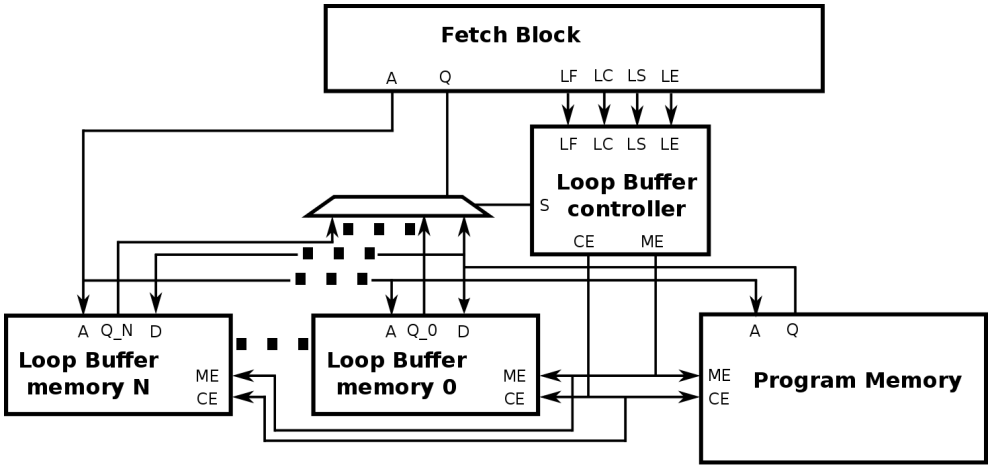


Figura A.15: Interfaz entre una arquitectura de procesador y una OMI.

A.5.2. Evaluación Experimental

A.5.2.1. Análisis de las Aplicaciones Experimentales

El consumo total de energía de los sistemas que se presentan en esta Sección está fuertemente influenciado por la OMI. Por un lado, Figura A.16, Figura A.17, Figura A.18, y Figura A.19 presentan el primer resultado de esta evaluación experimental. En estas Figuras, los componentes de la arquitectura de procesador están agrupados. Aparte de ver los cambios en la distribución de energía de un diseño basado en un procesador de propósito general a uno basado en un diseño ASIP, estas Figuras demuestran que el consumo total de energía de estos sistemas está fuertemente influenciado por la OMI. Por otro lado, Figura A.20, Figura A.21, Figura A.22, y Figura A.23 muestran información de los accesos que son realizados en el espacio de direcciones de la memoria de programa. En estas Figuras se puede ver que hay regiones que son más frecuentemente accedidas que otras, lo que implica la existencia de bucles. Aparte de este hecho, también se puede ver que los tiempos de ejecución de las aplicaciones seleccionadas están dominados por sólo unos pocos bucles.

Para implementar OMIs eficientes energéticamente basándose en arquitecturas de *loop buffer*, más información sobre bucles es necesitada. Tabla A.4, Tabla A.5, Tabla A.6 y Tabla A.7 proporcionan esta información. En estas Tablas, los bucles están numerados en el mismo orden en que aparecen en el código ensamblador del algoritmo. Un bucle anidado crea otro nivel de numeración. Así, un bucle denominado *2* corresponde al segundo bucle encontrado, mientras que un bucle denominado *2.1* se corresponde con el primer sub-bucle encontrado en el bucle denominado *2*. Estas Tablas corroboran el hecho de que el tiempo de ejecución de los bucles domina el tiempo total de ejecución de la aplicación. Por ejemplo, el tiempo de ejecución de los bucles en el algoritmo HBD representa aproximadamente un 79 % del tiempo de ejecución total en el caso del procesador de propósito general, y un 81 % en el procesador que está optimizado para este algoritmo. Por el contrario, en el algoritmo AES, el tiempo de ejecución de los bucles representa un 77 % del tiempo total de ejecución en el caso del procesador de propósito general, y un 90 % en el procesador que está optimizado para este algoritmo. Es necesario señalar que existen diferencias entre algoritmos de la misma aplicación, porque, al código fuente de los algoritmos optimizados, se le ha aplicado transformaciones y optimizaciones a fin de generar los códigos eficientes.

Las configuraciones de las arquitecturas CELB y BCLB se analizan en esta Sección basándose en los perfiles de bucle presentados en Tabla A.4, Tabla A.5, Tabla A.6 y Tabla A.7. Por un lado, la selección de las configuraciones de la arquitectura CELB se basa en tomar el menor tamaño de bucle que tiene mayor porcentaje de tiempo de ejecución. Con esta estrategia, son seleccionadas

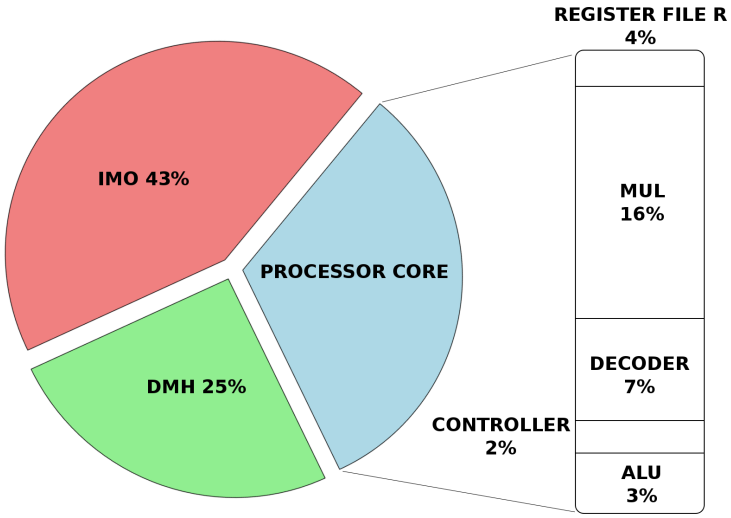


Figura A.16: Desglose de potencia en el procesador de propósito general ejecutando el algoritmo HBD.

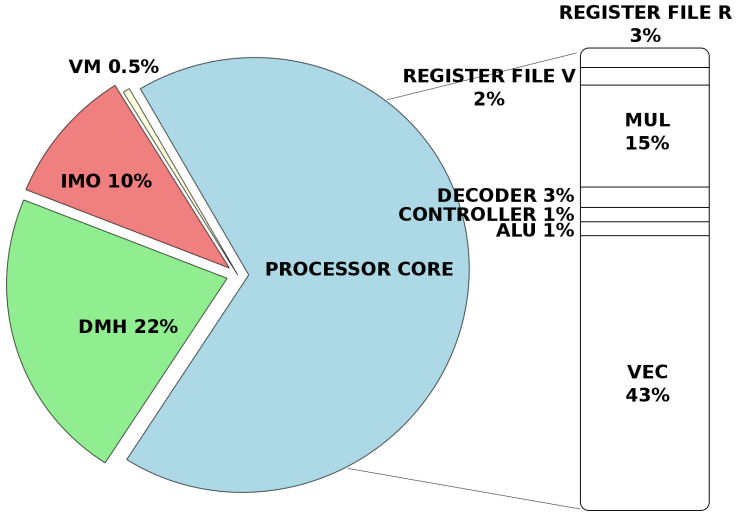


Figura A.17: Desglose de potencia en el procesador optimizado ejecutando el algoritmo HBD.

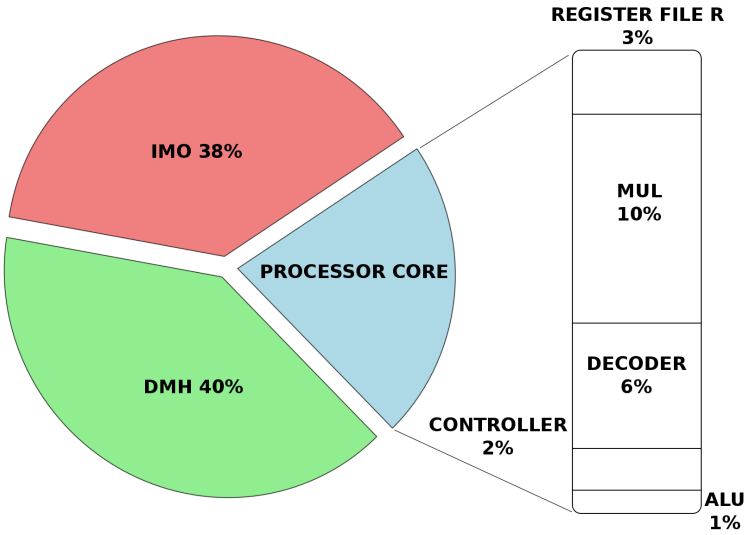


Figura A.18: Desglose de potencia en el procesador de propósito general ejecutando el algoritmo AES.

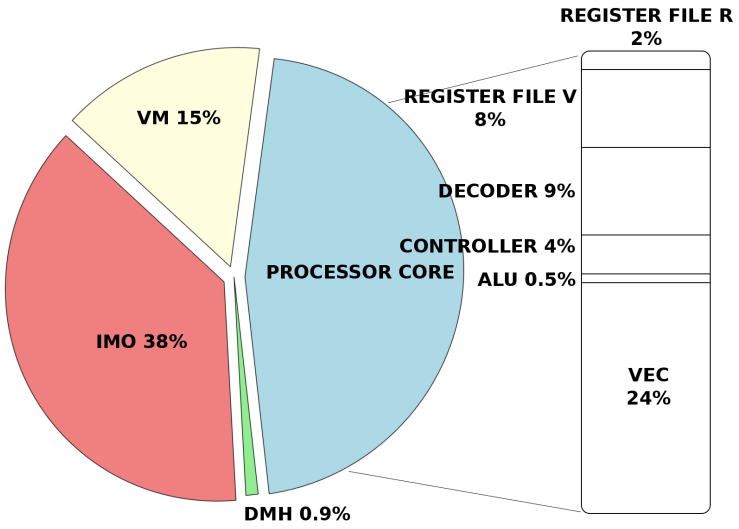


Figura A.19: Desglose de potencia en el procesador optimizado ejecutando el algoritmo AES.

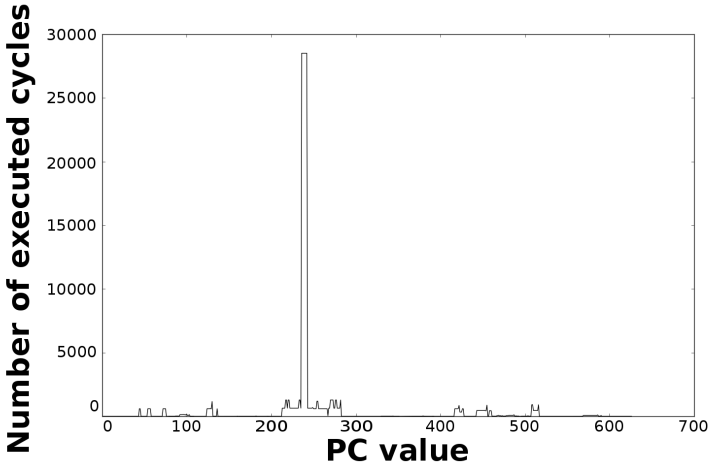


Figura A.20: Número de ciclos por PC en el procesador de propósito general ejecutando el algoritmo HBD.

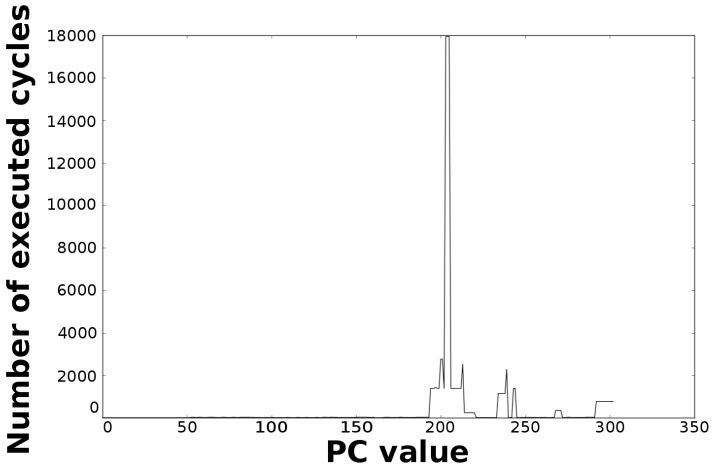


Figura A.21: Número de ciclos por PC en el procesador optimizado ejecutando el algoritmo HBD.

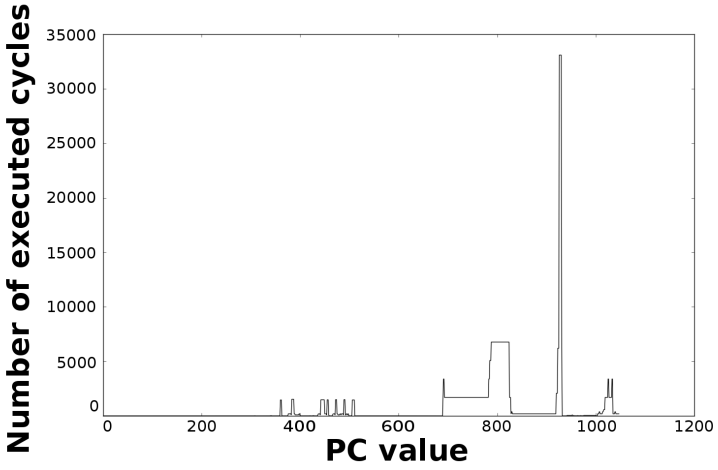


Figura A.22: Número de ciclos por PC en el procesador de propósito general ejecutando el algoritmo AES.

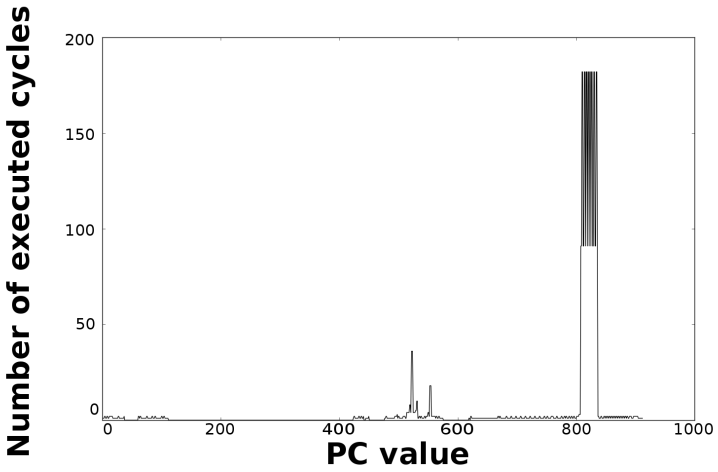


Figura A.23: Número de ciclos por PC en el procesador optimizado ejecutando el algoritmo AES.

las configuraciones más eficientes energéticamente. Estos ahorros de energía ayudan a reducir la penalización energética relacionada con la introducción de la arquitectura de *loop buffer*. Por otro lado, la selección de las configuraciones de la arquitectura BCLB se basa en la estrategia de tomar el mayor tamaño de bucle existente en la aplicación, y dividirlo por la granularidad del tamaño de bucle más pequeño contenido en la aplicación. Esta estrategia es utilizada en estas arquitecturas, debido a que el consumo exacto de energía de la lógica adicional que tiene que ser añadida al controlador de la OMI es desconocida. La Tabla A.8 presenta las configuraciones iniciales que son evaluadas.

Para concluir, es necesario remarcar que se ha utilizado una frecuencia en el sistema de 100MHz debido a los requisitos de tiempo impuestos por las aplicaciones. A esta frecuencia, el algoritmo HBD ejecutado en el procesador de propósito general usa 462 ciclos para examinar una muestra de entrada. Sin embargo, si este algoritmo es ejecutado en el procesador optimizado para este algoritmo, el número de ciclos para procesar una muestra de entrada es de 11 ciclos. Por otra parte, el algoritmo AES ejecutado en el procesador de propósito general usa 484 ciclos para procesar una muestra de entrada. Si este algoritmo es ejecutado en el procesador optimizado para este algoritmo, el número de ciclos para procesar una muestra de entrada es de sólo 3 ciclos.

Tabla A.4: Información de bucles del algoritmo HBD en el procesador de propósito general.

	Start address	**End address**	**Loop body size**	**Number of iterations**	**Execution time [%]**
Loop 1	33	34	2	4	0
Loop 2	44	45	2	594	0
Loop 3	54	57	4	594	1
Loop 4	72	75	4	594	1
Loop 5	92	103	12	132	1
Loop 6	124	136	13	594	3
Loop 7	160	160	1	15	0
Loop 8	236	242	7	32,625	71
Loop 9	417	427	11	594	2
Loop 10	569	590	22	64	0

Tabla A.5: Información de bucles del algoritmo HBD en el procesador optimizado.

	Start address	**End address**	**Loop body size**	**Number of iterations**	**Execution time [%]**
Loop 1	192	244	53	1,380	70
Loop 1.1	200	205	6	1	0
Loop 2	266	271	6	350	2
Loop 3	209	302	13	768	9

Tabla A.6: Información de bucles del algoritmo AES en el procesador de propósito general.

	Start address	End address	Loop body size	Number of iterations	Execution time [%]
Loop 1	307	309	3	8	0
Loop 2	324	327	4	2	0
Loop 3	340	342	3	16	0
Loop 4	360	362	3	1,460	3
Loop 5	383	387	5	1,600	7
Loop 6	409	411	3	4	0
Loop 7	419	421	3	8	0
Loop 8	426	428	3	16	0
Loop 9	436	458	23	92	2
Loop 10	472	474	3	1,392	3
Loop 11	489	491	3	1,392	3
Loop 12	506	510	5	1,460	6
Loop 13	519	523	5	4	0
Loop 14	926	930	5	6,016	25
Loop 15	942	1,000	59	40	2
Loop 16	1,019	1,034	16	1,692	26

Tabla A.7: Información de bucles del algoritmo AES en el procesador optimizado.

	Start address	End address	Loop body size	Number of iterations	Execution time [%]
Loop 1	519	524	6	36	5
Loop 2	544	560	17	2	1
Loop 2.1	550	555	6	0	0
Loop 3	806	837	32	91	84

Tabla A.8: Configuraciones del marco experimental.

	Baseline architecture	CELB	BCLB
HBD algorithm General-purpose processor	No loop buffer architecture	8 words	8 banks of 8 words
HBD algorithm Optimised processor	No loop buffer architecture	64 words	8 banks of 8 words
AES algorithm General-purpose processor	No loop buffer architecture	8 words	4 banks of 8 words
AES algorithm Optimised processor	No loop buffer architecture	32 words	4 banks of 8 words

A.5.2.2. Análisis del Consumo de Potencia

Tabla A.9, Tabla A.10 y Tabla A.11 presentan los consumos de potencia dinámica, los consumos de potencia relacionada con las corrientes de fugas y los consumos de potencia total de todas las configuraciones que se presentan en la Tabla A.8. Como se puede ver, el consumo de potencia total de la OMI es la suma de lo que consumen los componentes que forman la OMI (es decir, el controlador de *loop buffer*, la memoria de *loop buffer* y la memoria de programa). Además, también se puede ver que los sistemas optimizados para las aplicaciones siempre consumen menos energía que los sistemas de propósito general. Por lo tanto, la introducción de las arquitecturas CELB y BCLB no afecta a esta tendencia de consumo de energía.

Tabla A.9: Consumo de potencia de la arquitectura de referencia.

	Component	Dynamic power	Leakage power	Total power
HBD algorithm	IMO	4.44×10^{-06}	0.91×10^{-09}	4.44×10^{-06}
-	LB Controller	0	0	0
General-purpose	LB Memory	0	0	0
processor	PM	4.44×10^{-06}	0.91×10^{-09}	4.44×10^{-06}
HBD algorithm	IMO	3.57×10^{-07}	8.46×10^{-11}	3.57×10^{-07}
-	LB Controller	0	0	0
Optimised	LB Memory	0	0	0
processor	PM	3.57×10^{-07}	8.46×10^{-11}	3.57×10^{-07}
AES algorithm	IMO	1.81×10^{-06}	4.32×10^{-10}	1.82×10^{-06}
-	LB Controller	0	0	0
General-purpose	LB Memory	0	0	0
processor	PM	1.81×10^{-06}	4.32×10^{-10}	1.82×10^{-06}
AES algorithm	IMO	1.20×10^{-06}	2.11×10^{-10}	1.20×10^{-06}
-	LB Controller	0	0	0
Optimised	LB Memory	0	0	0
processor	PM	1.20×10^{-06}	2.11×10^{-10}	1.20×10^{-06}

Se puede ver de la Tabla A.10 que existe una disminución en la potencia dinámica de las arquitecturas CELB y BCLB en relación con las arquitecturas de referencia. Esto es porque la mayoría de los instrucciones se obtienen de una memoria pequeña en lugar de la gran memoria que forma la memoria de programa. Las arquitecturas CELB tienen un aumento en la potencia relacionada con las corrientes de fugas en relación con las arquitecturas de referencia, debido a la introducción de la arquitectura de *loop buffer*. También es posible ver la importancia del controlador de la memoria de *loop buffer*, que representa desde un 5 % del consumo de energía de la OMI en el sistema donde el algoritmo AES es ejecutado sobre un procesador de propósito general, hasta un 30 % en el sistema en el que el algoritmo AES es ejecutado sobre el procesador optimizado para este algoritmo.

Tabla A.10: Consumo de potencia de la OMI basada en una arquitectura CELB.

	Component	Dynamic power	Leakage power	Total power
HBD algorithm	IMO	1.74×10^{-06}	1.14×10^{-09}	1.74×10^{-06}
-	LB Controller	2.55×10^{-07}	1.60×10^{-10}	2.55×10^{-07}
General-purpose	LB Memory	6.97×10^{-08}	6.60×10^{-11}	6.97×10^{-08}
processor	PM	1.41×10^{-06}	9.16×10^{-10}	1.41×10^{-06}
HBD algorithm	IMO	1.40×10^{-07}	1.77×10^{-10}	1.40×10^{-07}
-	LB Controller	3.71×10^{-08}	2.66×10^{-11}	3.71×10^{-08}
Optimised	LB Memory	5.76×10^{-08}	6.56×10^{-11}	5.76×10^{-08}
processor	PM	4.50×10^{-08}	8.46×10^{-11}	4.51×10^{-08}
AES algorithm	IMO	1.76×10^{-06}	5.25×10^{-10}	1.76×10^{-06}
-	LB Controller	1.03×10^{-07}	7.39×10^{-11}	1.03×10^{-07}
General-purpose	LB Memory	9.54×10^{-09}	2.68×10^{-11}	9.54×10^{-09}
processor	PM	1.65×10^{-06}	4.25×10^{-10}	1.65×10^{-06}
AES algorithm	IMO	8.32×10^{-07}	4.12×10^{-10}	8.36×10^{-07}
-	LB Controller	2.43×10^{-07}	7.53×10^{-11}	2.47×10^{-07}
Optimised	LB Memory	1.79×10^{-07}	1.29×10^{-10}	1.79×10^{-07}
processor	PM	4.10×10^{-07}	2.13×10^{-10}	4.10×10^{-07}

Tabla A.11: Consumo de potencia de la OMI basada en una arquitectura BCLB.

	Component	Dynamic power	Leakage power	Total power
HBD algorithm	IMO	1.97×10^{-06}	1.47×10^{-09}	1.97×10^{-06}
-	LB Controller	4.72×10^{-07}	3.95×10^{-10}	4.72×10^{-07}
General-purpose	LB Memory	8.73×10^{-08}	1.59×10^{-10}	8.73×10^{-08}
processor	PM	1.41×10^{-06}	9.16×10^{-10}	1.41×10^{-06}
HBD algorithm	IMO	1.64×10^{-07}	3.83×10^{-10}	1.65×10^{-07}
-	LB Controller	5.51×10^{-08}	1.40×10^{-10}	5.51×10^{-08}
Optimised	LB Memory	6.39×10^{-08}	1.58×10^{-10}	6.39×10^{-08}
processor	PM	4.50×10^{-08}	8.46×10^{-11}	4.51×10^{-08}
AES algorithm	IMO	1.90×10^{-06}	7.40×10^{-10}	1.90×10^{-06}
-	LB Controller	2.35×10^{-07}	2.72×10^{-10}	2.35×10^{-07}
General-purpose	LB Memory	1.46×10^{-08}	4.29×10^{-11}	1.46×10^{-08}
processor	PM	1.65×10^{-06}	4.25×10^{-10}	1.65×10^{-06}
AES algorithm	IMO	6.60×10^{-07}	4.30×10^{-10}	6.60×10^{-07}
-	LB Controller	5.20×10^{-08}	1.10×10^{-11}	5.20×10^{-08}
Optimised	LB Memory	1.98×10^{-07}	2.06×10^{-10}	1.98×10^{-07}
processor	PM	4.10×10^{-07}	2.13×10^{-10}	4.10×10^{-07}

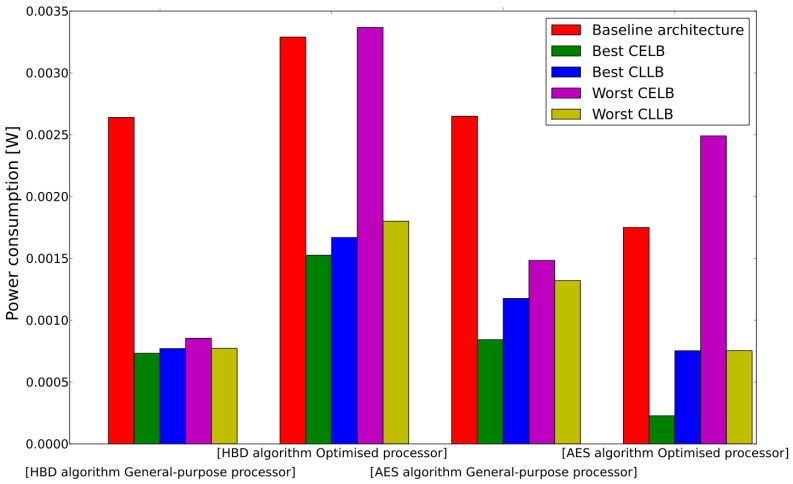


Figura A.24: Resumen de las mejores y peores arquitecturas CELB y BCLB.

En base a todos los resultados y todas las discusiones anteriores, es posible concluir que el uso de arquitecturas de *loop buffer* para optimizar la OMI desde el punto de vista de la eficiencia energética debe ser evaluado cuidadosamente. En los casos de estudio que se presentan en esta Sección, la arquitectura CELB normalmente es más eficiente energéticamente que la arquitectura BCLB, como puede verse en la Figura A.24. Sin embargo, no siempre es así. La mayor eficiencia energética de la arquitectura CELB se debe a que el tiempo de ejecución total de todos los *benchmarks* se concentra en unos bucles de tamaño similar. Si este tiempo de ejecución en un *benchmark* se encuentra compartido entre bucles con diferentes tamaños, la arquitectura BCLB nos traerá más eficiencia energética que la arquitectura CELB. Por lo tanto, los dos factores a tener en cuenta a fin de implementar una OMI eficiente energéticamente basada en una arquitectura de *loop buffer* son:

- **El porcentaje del tiempo de ejecución de la aplicación que está relacionado con la ejecución de bucles.** Si este porcentaje es bajo, la introducción de una arquitectura de *loop buffer* en la OMI no puede ofrecer ningún ahorro de energía, porque la arquitectura de *loop buffer* no se utiliza lo suficiente como para proporcionar los ahorros necesarios de energía. Por el contrario, cuanto mayor sea este porcentaje, mayor ahorro de energía se podrá lograr.
- **La distribución del porcentaje del tiempo de ejecución, que está relacionado con la ejecución de bucles, sobre cada uno de los bucles que forman la aplicación.** Si el tiempo total de ejecución se concentra en unos pocos bucles, la arquitectura CELB traerá más ahorros de energía que la arquitectura BCLB. Si este porcentaje se distribuye homogéneamente entre los bucles, la arquitectura BCLB traerá más ahorro de energía que la CELB. Estos datos se basan en el uso eficiente de los bancos múltiples que pueden formar la arquitectura de *loop buffer*.

A.6. Exploración del Espacio de Diseño de la Arquitectura DLB

Desde el punto de vista de un diseñador de sistemas empotrados, la OMI se convierte en un problema cuando las aplicaciones se ejecutan en arquitecturas VLIW, ya que típicamente en estas arquitecturas, la OMI se encuentra centralizada y tiene una baja eficiencia energética [JBA+05]. Debido a este hecho, el uso efectivo del paralelismo tiene para ser impulsado a fin de mejorar el rendimiento y la eficiencia energética en sistemas empotrados [Man05]. Por lo tanto, para este problema de consumo de energía se necesita una solución distribuida y escalable que utilice más de un hilo de ejecución y posea sobrecarga mínima de hardware.

En esta Sección, se proponen y analizan tres opciones de implementación de arquitecturas DLB eficientes para una aplicación determinada. Es importante aclarar que, para la OMI completa, el diseñador de sistemas empotrados tiene la opción no sólo de elegir una de las implementaciones propuestas para la lógica de control de la OMI, sino también de combinar estas tres implementaciones de la arquitectura DLB para lograr la configuración óptima de la OMI para una aplicación determinada. Las propuestas de implementación de la arquitectura DLB tienen en cuenta no sólo el posible ahorro de energía que se pueden conseguir en el sistema, sino también la ocupación de memoria y los rendimientos requeridos por parte de la aplicación empotrada.

A.6.1. Implementación de la Arquitectura DLB

Como se muestra en el ejemplo de motivación descrito en la Sección A.4.1, las arquitecturas convencionales de *loop buffer* (es decir, las arquitecturas CELB y CLLB) no son lo suficientemente eficientes energéticamente debido a los altos costes que estas arquitecturas muestran. Debido a este hecho, las arquitecturas DLB aparecen como una opción prometedora para mejorar la eficiencia energética de la OMI. En esta OMI existe particionamiento tanto en la arquitectura de *loop buffer* como en la memoria de programa. Como se muestra en la Figura A.25, las conexiones internas de esta OMI son gestionadas por una lógica de controladores que se distribuye a través de la arquitectura, ya que en esta arquitectura de *loop buffer* no sólo las memorias de *loop buffer* son distribuidas, sino también los controladores de bucle que las gestionan. Cada memoria de *loop buffer* tiene su propio controlador de bucle local. Debido a este hecho, la arquitectura DLB puede funcionar como una plataforma de múltiples hilos de ejecución, donde múltiples controladores sincronizables de bucle permiten la ejecución de múltiples bucles en paralelo. Sin embargo, la lógica del controlador de la arquitectura de *loop buffer* es simplificada y la sobrecarga en hardware es mínima, ya que tiene que ejecutar sólo código de

bucle. La Figura A.25 muestra la implementación de la arquitectura genérica de esta arquitectura de *loop buffer*. Como se muestra en esta Figura, esta arquitectura genérica está compuesta por el controlador y la memoria de la arquitectura de *loop buffer*.

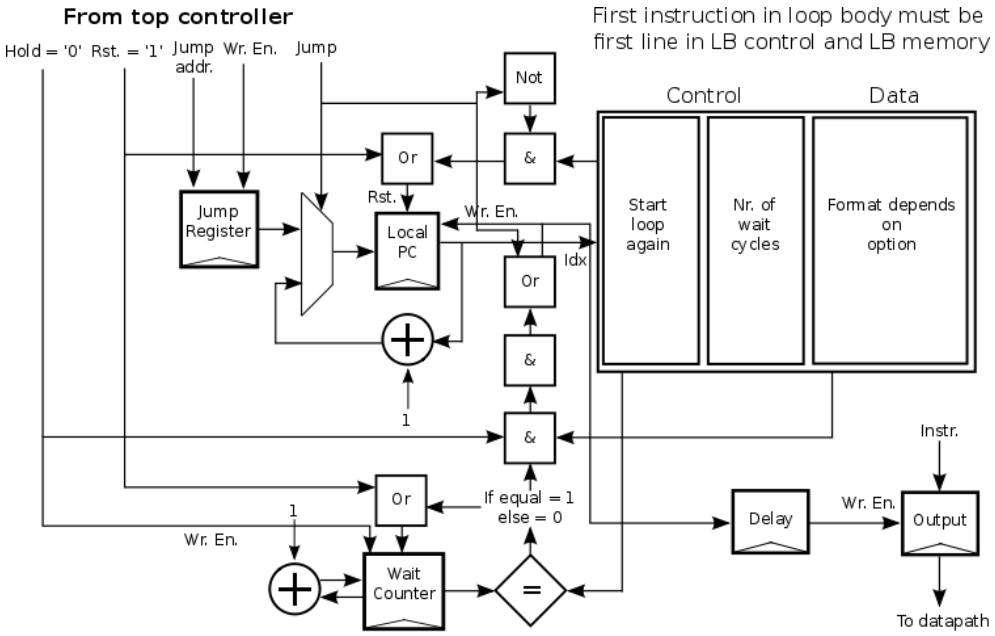


Figura A.25: Lógica y tabla de control del formato general de una arquitectura DLB.

Por un lado, la lógica circuital de control que se muestra en la Figura A.25 forma el controlador de la arquitectura de *loop buffer*, cuyos componentes principales son un registro de contador de programa local (*Local PC register*), un registro de salto (*JMP register*), un contador de espera (*Wait Counter register*) y una lógica de salida. El registro de Local PC almacena el contador de programa que se utiliza de forma local en la arquitectura de *loop buffer*. El registro de salto almacena la dirección donde el contador de programa local tiene que saltar. El registro del contador de espera contiene el número de ciclos que la arquitectura de *loop buffer* tiene que esperar para ser sincronizada. La lógica de salida es la parte del circuito que, basándose en el estado de los registros que forman el controlador de la arquitectura *loop buffer*, selecciona la instrucción que tiene que ser traída a la ruta de datos. Por otro lado, mientras que el controlador de la arquitectura *loop buffer* es común a todo el conjunto de implementaciones de la arquitectura DLB, la memoria de la arquitectura *loop buffer* puede ser implementada de diferentes maneras dependiendo de la opción que se haya seleccionado. Esta memoria es el bloque que tiene la etiqueta *Data* en la Figura A.25.

Esta Sección propone al diseñador de sistemas empotrados tres opciones prometedoras para la implementación de la arquitectura DLB. Estas opciones son complementarias y no solapadas, donde cada una es la más adecuada para una de las distintas particiones del espacio de diseño que está relacionado con la arquitectura DLB. Además, cada opción es claramente óptima para un patrón específico de código de la aplicación, el cuál se caracteriza por parámetros tales como el número de instrucciones diferentes, el tamaño de los bucles, la regularidad y el número de bits de cada instrucción. Con el fin de demostrar que estas opciones de implementación de la arquitectura DLB cubren todo el espacio de diseño de este tipo de OMI, cada opción es explicada en detalle en cada una de las siguientes Secciones.

A.6.1.1. Arquitectura DLB - OPCIÓN 1

La primera opción propuesta para la implementación de la arquitectura DLB está destinada específicamente a sistemas que presentan una secuencia regular y progresiva de obtención de instrucciones desde la arquitectura de *loop buffer*. Este tipo de conducta puede ser encontrada en unidades funcionales que requieran obtener instrucciones en un orden secuencial, como es el caso de la AG (*Address Generation Unit*).

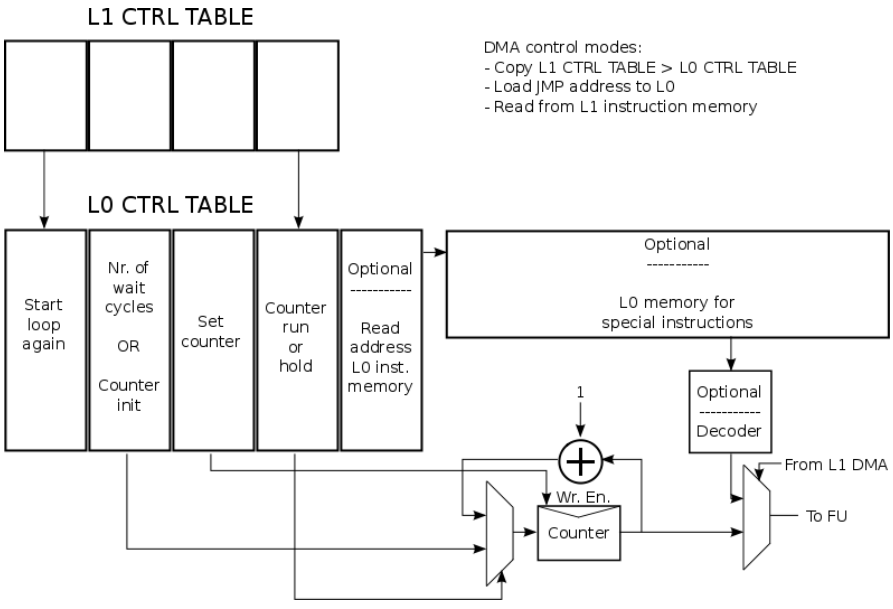


Figura A.26: Arquitectura DLB - OPCIÓN 1.

Como se muestra en la Figura A.26, la memoria de instrucción L0 se ha reemplazado por un contador. Este contador codifica las instrucciones

basándose en sus posiciones dentro del orden secuencial en las que son obtenidas. Por lo tanto, este contador es el elemento de la lógica circuital que controla qué instrucción está siendo ejecutada en cada unidad funcional asociada de la arquitectura de *loop buffer*. Este control se basa en la tabla de control, por lo que el tamaño de las instrucciones no afecta a cómo este contador es implementado. Debido a este hecho, la gran ventaja de esta opción es tener un coste global pequeño, por no aumentar el almacenamiento en la memoria L0 y requerir una lógica de control muy simple.

A.6.1.2. Arquitectura DLB - OPCIÓN 2

La segunda opción propuesta para la implementación de la arquitectura DLB es más adecuada cuando las instrucciones son obtenidas de la arquitectura de *loop buffer* en un orden no incremental. En esta Sección A.6.1.2 sólo se analiza el caso en donde las instrucciones son codificadas con un número pequeño de bits. El caso, en donde las instrucciones son codificadas con un número grande de bits, es analizado en Sección A.6.1.3.

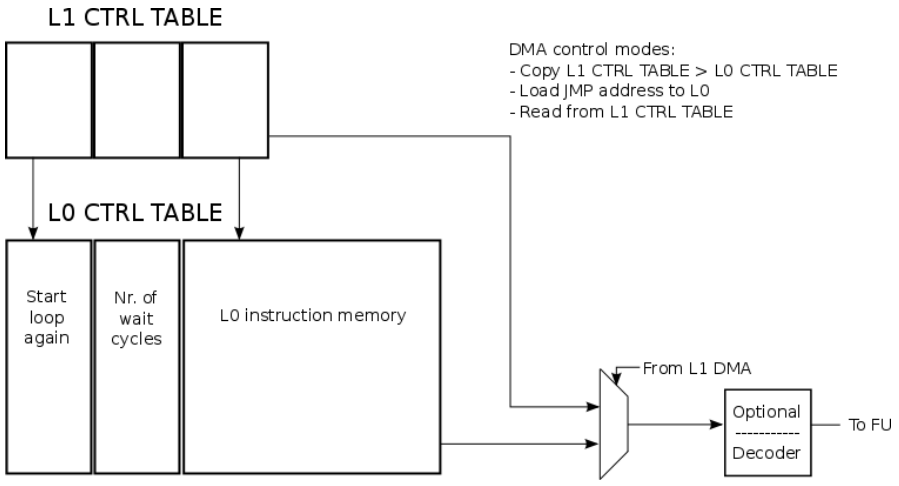


Figura A.27: Arquitectura DLB - OPCIÓN 2.

Como se muestra en la Figura A.27, en esta opción, la tabla de control es el elemento que controla la instrucción que es ejecutada en cada unidad funcional asociada de la arquitectura de *loop buffer*. Debido al hecho de que las instrucciones se encuentran codificadas en esta tabla de control, el tamaño de las instrucciones afecta al consumo de energía de la arquitectura de *loop buffer*. La ventaja de esta opción es que la arquitectura de *loop buffer* puede reducir en gran medida el consumo de energía de la OMI si el número de bits que se utiliza para codificar las instrucciones es pequeño.

A.6.1.3. Arquitectura DLB - OPCIÓN 3

La tercera opción propuesta es la más adecuada si las instrucciones son obtenidas desde la arquitectura de *loop buffer* en un orden no incremental, y éstas están codificadas con un gran número de bits, pero bajo la condición de que solamente exista unas pocas instrucciones distintas.

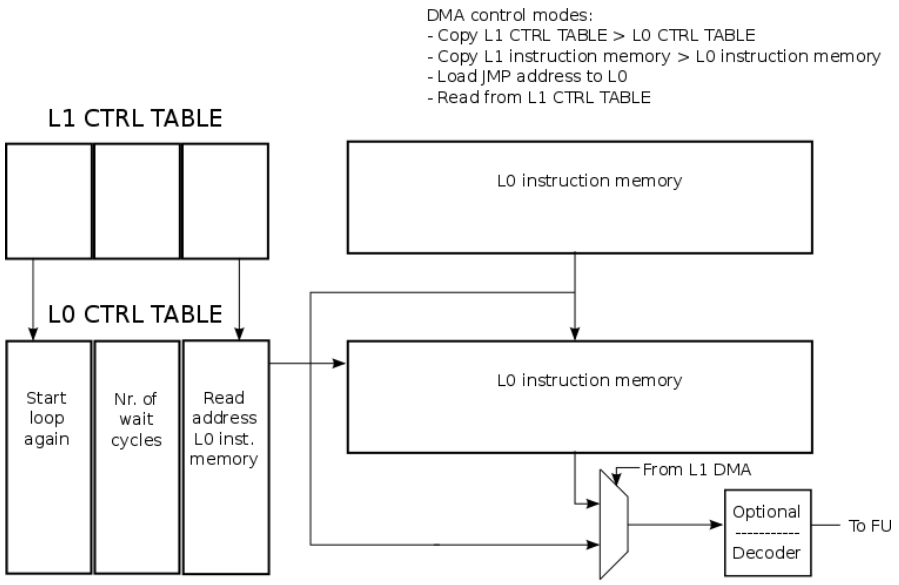


Figura A.28: Arquitectura DLB - OPCIÓN 3.

Como se muestra en la Figura A.28, la tabla de control es en este caso también el elemento de la lógica de control que controla qué instrucción es ejecutada en la unidad funcional asociada de la arquitectura de *loop buffer*. Sin embargo, debido al hecho de que las instrucciones son almacenadas en las propias memorias de la arquitectura de *loop buffer*, la tabla de control incrementa su complejidad a fin de controlar el flujo de accesos a estas memorias.

Con el fin de obtener ahorros de energía, esta opción tiene como requisito que las instrucciones que se utilizan en la memoria de la arquitectura de *loop buffer* estén codificadas con un gran número de bits. Sin embargo, pueden aparecer dos posibles escenarios dependiendo de cómo el cuerpo de un bucle es formado. Por un lado, si el cuerpo del bucle está formado por muchas instrucciones diferentes, la tabla de control almacena la dirección de la memoria de la arquitectura de *loop buffer* en la que cada instrucción se encuentra almacenada. Por otro lado, si el cuerpo del bucle está formado por pocas instrucciones diferentes, el campo que almacena la dirección de la memoria de la arquitectura de *loop buffer* en la que la instrucción es almacenada puede ser eliminado.

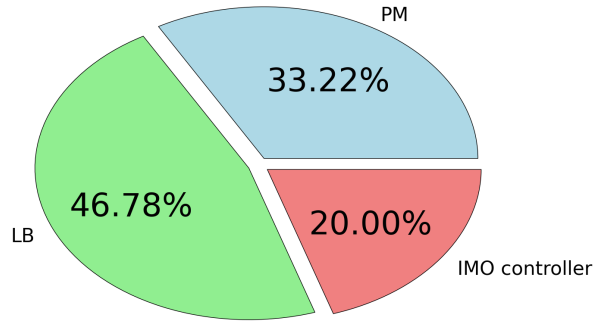
A.6.2. Resultados Experimentales

El marco experimental creado para los *benchmarks* que son presentados en esta Sección consta de una JMD, una OMI, una arquitectura de procesador y un interfaz de E/S. Las simulaciones energéticas, que se muestran en esta Sección, se han llevado a cabo utilizando la herramienta de estimación de energía de alto nivel descrita en la Sección A.3. Las herramientas de *Target Compiler Technologies* [TAR12] se han utilizado para corroborar el correcto funcionamiento de los sistemas, así como generar el informe histórico de accesos a la memorias (ver Apéndice B). En este marco experimental, la arquitectura de *loop buffer* consta de una memoria y un controlador. La implementación de la memoria de la arquitectura de *loop buffer* se basa en un conjunto de bancos de memoria, en el que cada banco puede ser configurado para ajustarse al tamaño deseado. El controlador de la arquitectura de *loop buffer* es el componente que controla el estado de la memoria de la arquitectura de *loop buffer* dentro de la OMI. La evaluación se ha realizado usando memorias comerciales y librerías LP de 90nm de TSMC, y fijando la frecuencia de reloj a 100MHz.

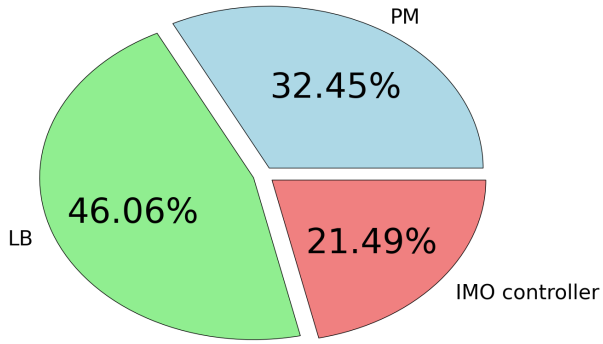
En primer lugar, esta Sección muestra cómo las arquitecturas DLB sacrifican una parte del consumo de energía de la lógica de control de la OMI a fin de ahorrar energía. La disyuntiva entre el consumo de energía de la lógica de control y los posibles ahorros de energía en la OMI se muestran en la Sección A.6.2.1. En segundo lugar, las opciones para la implementación de la arquitectura DLB que se proponen en esta Sección son analizadas con varios casos de estudio. El análisis de alto nivel de las disyuntivas en el consumo de energía de la exploración del espacio de diseño de la arquitectura DLB requiere *benchmarks* con patrones diferentes en constitución y ejecución. Por un lado, la Sección A.6.2.2 describe los *benchmarks* sintéticos que se han desarrollado para mostrar las tendencias en el consumo de energía de cada una de las diferentes implementaciones arquitecturales propuestas en la Sección A.6.1. Por otro lado, la Sección A.6.2.3 presenta la evaluación y el análisis de las implementaciones propuestas basándose en aplicaciones empotradas de la vida real para mostrar su tendencia en el consumo de energía.

A.6.2.1. Comparación con Soluciones Convencionales

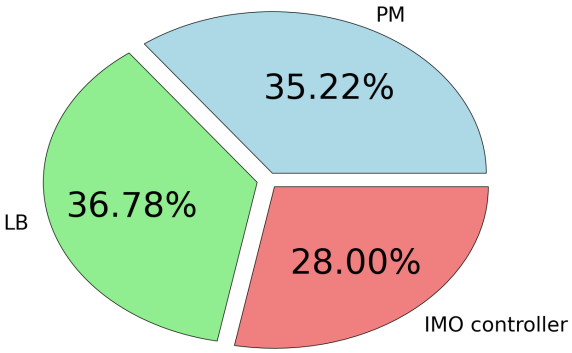
El *benchmark AES NON-OPTIMISED* que se presenta en la Tabla A.12 es utilizado en la presente Sección. La Figura A.29 muestra cómo para este *benchmark* la distribución del consumo de energía en la OMI cambia de una arquitectura representativa a otra. Si las arquitecturas CELB y CLLB se comparan, es posible ver que la energía relacionada con el controlador de la arquitectura de *loop buffer* es incrementada en la arquitectura CLLB con



(a) Arquitectura CELB.



(b) Arquitectura CLLB.



(c) Arquitectura DLB.

Figura A.29: Desglose de energía de diferentes arquitecturas de *loop buffer* ejecutando el *benchmark AES NON-OPTIMISED*.

respecto a la arquitectura CELB, debido al hecho de que el controlador tiene que controlar la activación o desactivación de más componentes. Además, es posible apreciar cómo el consumo de la memoria de la arquitectura de *loop buffer* se reduce. Este hecho está relacionado con la reducción en el consumo de energía dinámica que está causada por el uso de instancias de memoria más pequeñas. Si la arquitectura DLB es analizada, se puede apreciar que el porcentaje del consumo de energía que está relacionado con el controlador de esta arquitectura es mayor que el de los controladores de las arquitecturas de *loop buffer* anteriores. Esto es debido al hecho de que el controlador de esta arquitectura de *loop buffer* es más complejo. Sin embargo, como se puede ver de esta Figura A.29, la gestión eficiente que es realizada por el controlador lleva a mayores reducciones en la arquitectura de *loop buffer* y como consecuencia en la energía total de la OMI. Hay que señalar que el valor absoluto del consumo de energía de la memoria de programa no es una constante en todas estas OMIs debido a dos hechos. En primer lugar, el número de ciclos que la aplicación requiere para su ejecución a través de estas arquitecturas cambia. En segundo lugar, el tamaño de la memoria cambia, así como su composición cuando se encuentra basada en bancos de memoria. La variación en el tamaño de las memorias que forman la arquitectura de *loop buffer* afecta más a la arquitectura CELB que a las arquitecturas CLLB o DLB, ya que la gestión eficiente que la lógica de control de estas arquitecturas tienen para la activación o desactivación de las instancias de memoria, compensa la posible sobrecarga que puede ser creada por la variación de las instancias de memoria que forman la memoria de la arquitectura de *loop buffer*.

A.6.2.2. *Benchmarks* Sintéticos

Para ver los ahorros potenciales de energía de cada una de las implementaciones propuestas en esta Sección, *benchmarks* con patrones específicos (ver Tabla A.12) tienen que ser utilizados para mostrar claramente el caso especial en el que cada opción es óptima. Estos *benchmarks* imitan bucles que pueden ser encontrados en aplicaciones empotradas de la vida real. Cada bucle que está incluido en los *benchmarks* sintéticos es caracterizado basándose en dos parámetros: número de iteraciones y tamaño del bucle. Los tamaños de los bucles en los *benchmarks* sintéticos se muestran en la Figura A.9, donde se puede ver que cada *benchmark* está compuesto de tres bucles de 4, 16 y 32 instrucciones respectivamente. Con el fin de controlar con suficiente precisión los tamaños de los bucles, los *benchmarks* sintéticos son implementados en código ensamblador. Las instrucciones y los operandos que componen cada bucle son asignados al azar. Ésto es diferente de la realidad, donde alguna correlación está presente entre los bits de instrucción, pero para el propósito de este experimento, estas correlaciones no son tan importantes, ya que tienen bajo impacto en el consumo de energía de la OMI.

Tabla A.12: Aplicaciones empotradas sintéticas y reales usadas como *benchmarks*.

Synthetic Benchmark (See Section A.6.2.2)	Cycles	Issue slots	Bits per instruction	LB size [Instructions]	Loop code [%]	NOP instructions [%]
SB 1	3,050	2	16	32	96.88	0.07
SB 2	3,050	2	16	32	96.88	0.07
SB 3	3,050	2	32	32	96.88	0.07
Real-life Benchmark [Reference]	**Cycles**	**Issue slots**	**Bits per instruction**	**LB size [Instructions]**	**Loop code [%]**	**NOP instructions [%]**
AES NON-OPTIMISED [TSH+10] (See Section C.2)	707,052	1	16	64	1.68	8.03
AES OPTIMISED [TSH+10] (See Section C.3)	3,347	1	16	32	77.44	0.09
BIO-IMAGING [PFH+12] (See Section C.4)	334,071	4	80	64	98.01	26.70
CWT NON-OPTIMISED [YKH+09] (See Section C.5)	274,464	1	16	32	6.61	0.48
CWT OPTIMISED [YKH+09] (See Section C.6)	102,827	1	20	64	6.36	2.50
DWT NON-OPTIMISED [DS98] (See Section C.7)	758,216	2	16	518	65.70	25.19
DWT OPTIMISED [DS98] (See Section C.8)	317,739	2	32	4	1.89	1.73
MRFA [QJ04] (See Section C.9)	177,170	2	32	64	19.13	17.01

Los tres *benchmarks* sintéticos que se utilizan en esta Sección se describen en los siguientes puntos:

SB 1 Este *benchmark* sintético tiene la intención de mostrar que la opción número 1 para implementar la arquitectura DLB es la más eficiente energéticamente, siempre que el sistema presente una secuencia de obtención de instrucción regular y progresiva. Por lo tanto, en base a esta premisa, *SB 1* se compone de bucles con instrucciones que son obtenidas de la arquitectura de *loop buffer* en orden secuencial.

SB 2 La ventaja de la opción número 2 es que la arquitectura de *loop buffer* puede reducir considerablemente el consumo de energía de la OMI, si el número de bits que se utilizan para codificar las instrucciones es pequeño. Debido al hecho de que, en esta opción los diseñadores de sistemas no tienen que tener en cuenta el orden de obtención de instrucciones de la unidad funcional asociada que sigue, *SB 2* es implementado con el mismo número de bits por instrucción que *SB 1*. Sin embargo, en este caso, los bucles que forman este *benchmark* presentan orden secuencial no regular cuando las instrucciones son obtenidas de la arquitectura de *loop buffer*. Las instrucciones que forman cada bucle de este *benchmark* son asignadas al azar con una probabilidad de cambio del 50 % en relación con el patrón de las instrucciones contenidas en *SB 1*.

SB 3 Este último *benchmark* sintético está destinado a mostrar el caso en el que la opción más eficiente de energía es la número 3. En este *benchmark* sintético, los bucles que forman el *benchmark* presentan no sólo un orden secuencial no regular de obtención de instrucciones, sino también un número de bits por instrucción mayor (ancho de palabra de la instrucción) en comparación con los *benchmarks SB 1* y *SB 2*.

En la Figura A.30, las implementaciones propuestas son comparadas con la arquitectura CELB. Como puede verse, cuando se utiliza *SB 1*, la opción más eficiente es la número 1. Como *SB 1* es un *benchmark* con un número pequeño de bits por instrucción, la opción de implementación número 2 es mejor que la opción número 3. De los resultados de *SB 2*, es posible ver que en este caso, la mejor opción de implementación no es la número 1, sino la número 2. Además, la opción número 1 es mejor que la opción número 3, porque el número de bits por instrucción introduce una mayor sobrecarga en el consumo de energía que la penalización por el orden secuencial no regular de obtención de instrucciones. Como se menciona anteriormente, el *benchmark SB 3* es el mismo que el *SB 2*, pero el primero tiene más bits por instrucción. Debido a este hecho, cuando *SB 3* es ejecutado a través de las diferentes opciones, sólo la opción número 3 es eficiente energéticamente. La baja sobrecarga en la lógica de control que presenta la opción número 1 en comparación con la opción número 2 hace que la primera opción sea más eficiente energéticamente cuando se utiliza *SB 3*. Desde el punto de vista de la opción número 1, el mejor

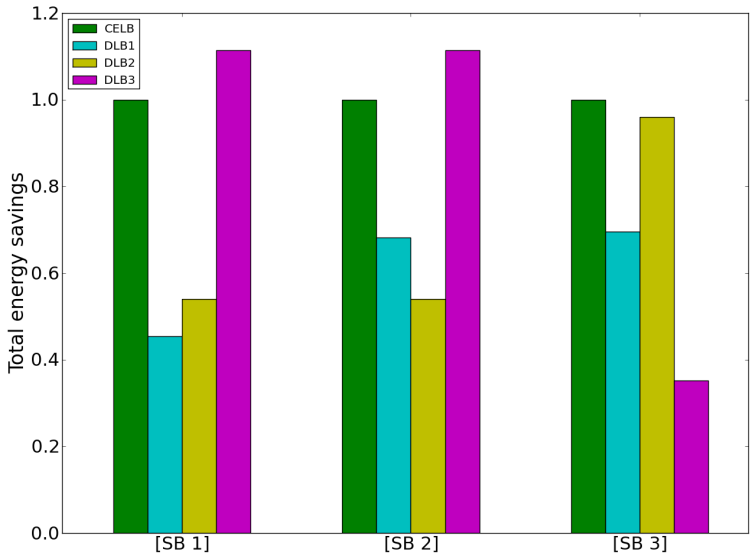


Figura A.30: Consumo de energía normalizado en las arquitecturas DLB usando los *benchmarks* sintéticos.

benchmark sintético es *SB 1*. Sin embargo, es posible ver que no hay una gran diferencia entre la ejecución de *SB 2* y *SB 3*, ya que para esta implementación, estos *benchmarks* presentan el mismo patrón de ejecución. Desde el punto de vista de la opción número 2, el mejor *benchmark* sintético es o *SB 1* o *SB 2*, ya que ambos tienen un pequeño número de bits por instrucción. *SB 3* no es bueno para esta implementación, ya que este *benchmark* tiene una mayor número de bits por instrucción en comparación con *SB 1* y *SB 2*. Desde punto de vista de la opción número 3, sólo el *benchmark SB 3* es bueno debido al número de bits por instrucción. La secuencia regular o no en la obtención de instrucciones no es importante para esta opción, ya que es posible ver una pequeña diferencia entre la ejecución de los *benchmarks SB 1* y *SB 2*. La conclusión de esta Sección es que existen patrones específicos, que se pueden encontrar en el código de la aplicación, que pueden ayudar a los diseñadores de sistemas empotrados a elegir la implementación óptima de la arquitectura DLB desde el punto de vista del consumo de energía. Este hecho es explotado en las Secciones siguientes.

A.6.2.3. *Benchmarks* Reales

La Figura A.31 muestra que es posible aumentar un 6 %– 22 % el ahorro de energía de la arquitectura de *loop buffer*. Sin embargo, para ello se requiere un aumento medio del consumo de energía de 6.5 % de la lógica de control.

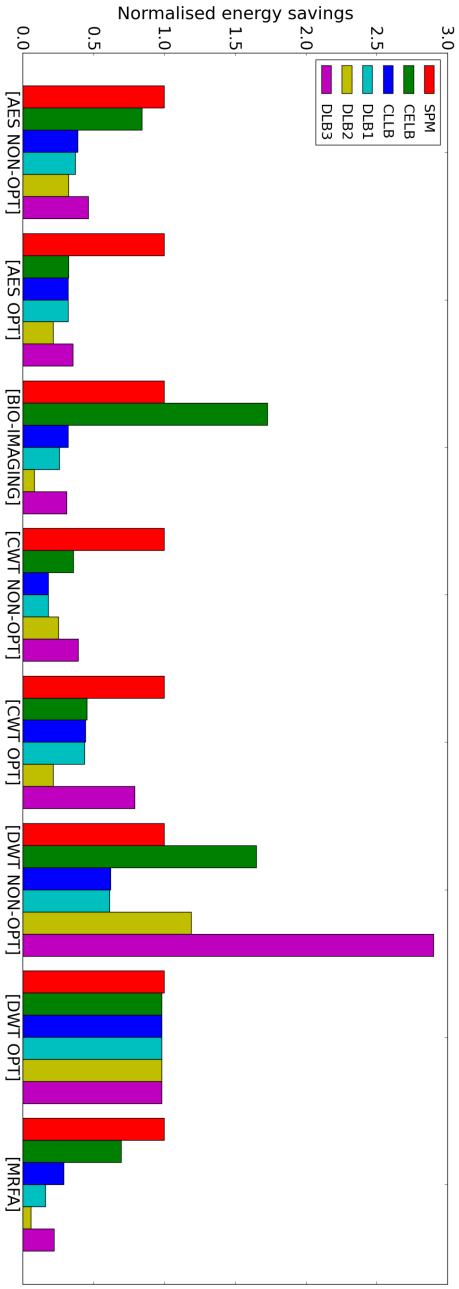


Figura A.31: Ahorros de energía normalizados en diferentes OMIs.

La Tabla A.12 muestra los *benchmarks* basados en aplicaciones empotradas de la vida real utilizados en esta Sección. Este conjunto de *benchmarks* está formado por aplicaciones empotradas que son ejemplos excelentes, no sólo de todos los dominios de aplicación que se encuentran dominados por bucles y exhiben suficiente oportunidad de paralelismo a nivel de datos y/o de instrucción, sino también de dominios de aplicaciones de propósito más genérico que pueden ser encontrados en los diferentes tipos de procesamiento de la señal. Esta selección de *benchmarks* se realizó no sólo para completar el análisis de alto nivel presentado en esta Sección, sino también para hacer que el análisis que se presenta en esta Sección sea lo suficientemente genérico, a fin de ser aplicable a todos los dominios de aplicación de los sistemas empotrados que se encuentran dominados por bucles.

La Figura A.31 muestra los ahorros de energía normalizados en escala logarítmica de cada una de las implementaciones propuestas para la arquitectura DLB cuando los *benchmarks* reales son ejecutados sobre estas implementaciones. Como se puede ver en esta Figura, la opción de implementación número 1 es la mejor elección para muchos de los *benchmarks* reales. El grado de diferencia entre la opción número 1 y las otras dos opciones de implementación depende de la secuencia con la que se obtienen las instrucciones de la arquitectura de *loop buffer*. Este hecho se puede apreciar entre las implementaciones optimizadas de los *benchmarks* reales y las implementaciones no optimizadas de ellos. Desde el punto de vista de la opción de implementación número 2, el *benchmark DWT OPTIMISE* es el que claramente aprovecha las ventajas de esta implementación. Esto es debido al pequeño número y tamaño de instrucciones que tienen que ser almacenadas en la tabla de control de la arquitectura de *loop buffer*. Desde el punto de vista de la opción de implementación número 3, los *benchmarks CWT NON-OPTIMISED* y *DWT NON-OPTIMISED* son los que se aprovechan de los ahorros de energía que ofrece esta implementación. Estos *benchmarks* presentan una secuencia no regular de obtención de instrucciones. Además, el número de bits por instrucción y el número de instrucciones que tienen que ser almacenados en estos *benchmarks* son ambos altos. Se puede ver varias anomalías a lo largo de los resultados experimentales obtenidos de los *benchmarks* reales. La primera anomalía se encuentra en el *benchmark BIO-IMAGING*. Esta anomalía está relacionada con el gran tamaño de la instancia de memoria que se requiere por la arquitectura CELB para ejecutar este *benchmark*. Para este *benchmark* real, durante partes del código de bucle de la aplicación, una única instancia de memoria tiene que almacenar 64 instrucciones de 80 bits cada una. Esta instancia grande de memoria está completamente activa cuando sólo es accedida una única dirección. Este hecho aumenta mucho el consumo total de energía de la OMI. El resto de las arquitecturas de *loop buffer* tienen más instancias de memoria que son más pequeñas y pueden estar activadas o no al mismo tiempo. En el caso de la

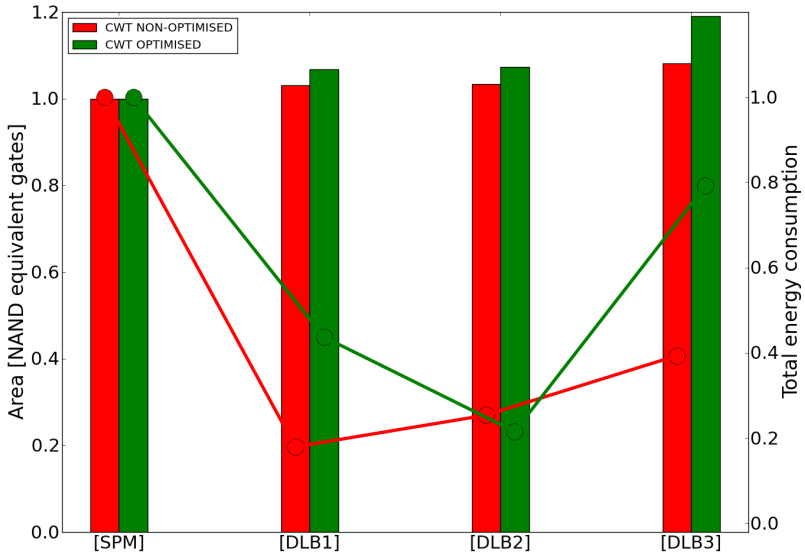


Figura A.32: Consumo de energía normalizado (representado por lineas) vs. ocupación en área (representada por barras) en diferentes OMIs.

anomalía relacionada con el *benchmark DWT NON-OPTIMISED*, se puede ver que éste posee el mismo problema que el *benchmark BIO-IMAGING*. Sin embargo, en este caso, las instancias de memoria son más grandes (ver Tabla A.12). Este problema es mayor en la opción número 2 que en la arquitectura CELB, porque en la opción número 2 la instancia de memoria no sólo contiene el código de instrucción, sino también los bits para el control de la arquitectura de *loop buffer*. En el caso de la arquitectura CELB, la instancia de memoria sólo tiene que almacenar las instrucciones en una única instancia grande de memoria.

La Figura A.32, basada en curvas de Pareto que relacionan el consumo de energía y la ocupación de área, muestra la sobrecarga en la ocupación de área de memoria que se requiere para lograr ahorros de energía en dos de los *benchmarks* de la Tabla A.12. Como se muestra en esta Figura, la penalización en área que el diseñador de sistemas empotrados tiene que asumir a fin de lograr mayor ahorro de energía es relativamente pequeño (un 10 % en promedio). Pero, esto aún muestra una interesante disyuntiva que tiene que decirse en base al contexto global del diseño, donde las características de la aplicación y los requerimientos del sistema deben ser tenidos en cuenta.

De los resultados que se han presentado a lo largo de esta Sección, se puede concluir que cada opción de implementación para la arquitectura DLB es claramente óptima para una patrón específico de código de aplicación. Debido a este hecho, la OMI óptima, desde el punto de vista de la eficiencia energética, tiene que combinar estas opciones de implementación complementarias y no solapadas para así cubrir las distintas particiones del espacio de diseño de la arquitectura DLB. Si el sistema empotrado es diseñado solamente para una sola aplicación específica, el diseñador de sistemas empotrados puede fijar los parámetros para conseguir la configuración óptima específica que crea la implementación más eficiente de la OMI para el código de aplicación dado. Sin embargo, si hay un conjunto de aplicaciones que el usuario desea ejecutar en el sistema empotrado, en este caso, este sistema tiene que utilizar reconfiguración dinámica basada en información dinámica del perfil de aplicación, a fin de estar adaptada a todas las aplicaciones que el usuario desea ejecutar en el sistema empotrado. De este modo, el sistema utiliza la implementación más eficiente de la arquitectura DLB para cada parte específica del código de aplicación que coincide con cada uno de los patrones mostrados en Sección A.6.2.2. La Figura A.33 muestra un ejemplo de sistema empotrado con las opciones de implementación complementarias y no solapadas de la arquitectura DLB propuestas en esta Sección. Si debido a problemas en el diseño del sistema empotrado, una sola opción de las implementaciones propuestas de la arquitectura DLB tiene que ser elegida, las directrices para el uso de cada opción, basadas en características de la aplicación tales como el número de instrucciones diferentes, el tamaño de los bucles, la regularidad en la obtención de las instrucciones y el número de bits utilizados en la codificación de cada instrucción, son:

- **OPCIÓN 1.** El código de la aplicación presenta una secuencia regular en la obtención de instrucciones de la arquitectura de *loop buffer*. El número de bits utilizados en la codificación de las instrucciones de la aplicación no afecta a la eficiencia energética de esta implementación de la arquitectura DLB.

- **OPCIÓN 2.** El código de la aplicación presenta instrucciones codificadas con un número pequeño de bits. La secuencia en la que se obtiene la instrucción no afecta a la eficiencia energética de esta implementación de la arquitectura DLB.

- **OPCIÓN 3.** El código de la aplicación presenta instrucciones con un gran número de bits en su codificación. La secuencia en la que se obtienen las instrucciones no afecta a la eficiencia energética de esta implementación de la arquitectura DLB.

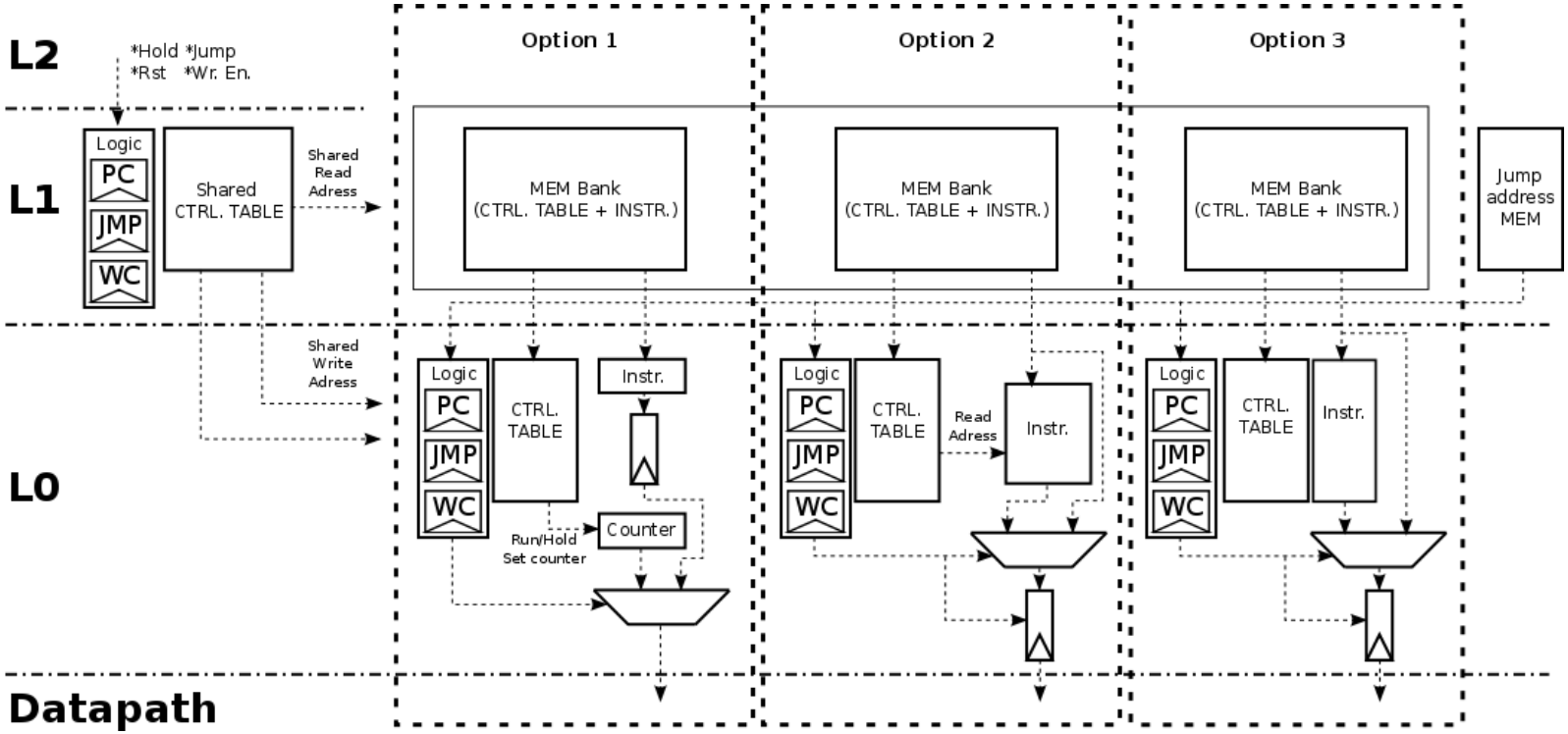


Figura A.33: Ejemplo de un sistema empotrado con las tres opciones de implementación de la arquitectura DLB.

A.7. Conclusiones y Trabajo Futuro

Como se muestra en la Sección A.1.1, los sistemas empotrados no sólo poseen un tamaño de mercado que es sobre 100 veces el mercado de los ordenadores, sino también ofrecen la más amplia variedad de potencia de procesamiento y de coste dentro del mercado de la informática. Sin embargo, a pesar de esta variedad, todo sistema empotrado se encuentra limitado por las siguientes cuestiones: el rendimiento requerido, la optimización del área ocupada por la memoria y la reducción del consumo de energía. Entre estas tres cuestiones, la optimización del consumo de energía esta llegando a ser crucial en el diseño de sistemas empotrados eficientes, no sólo por la demanda creciente de sistemas alimentados por baterías, sino también por la necesidad de usar empaquetados menos costosos. Debido a este hecho, los diseñadores de sistemas empotrados tienen que mirar al sistema en su conjunto y abordar el problema del consumo de energía en cada uno de los componentes que forman el sistema.

Investigaciones previas, como se puede ver en la Sección A.2, han señalado la organización de la memoria de instrucciones como una de las fuentes mayores de consumo de energía de los sistemas empotrados. Esta tesis doctoral ha introducido el estudio, el análisis, la propuesta, la implementación, y la evaluación de técnicas de optimización para bajo consumo de energía que pueden ser usadas en las organizaciones de la memoria de instrucciones de los sistemas empotrados:

- El Capítulo 1 ha presentado la motivación y el contexto del problema relacionado con el consumo de energía de la organización de la memoria de instrucciones en sistemas empotrados.

- El Capítulo 2 ha proporcionado un estudio completo de la literatura y una visión del estatus actual de las técnicas de bajo consumo de energía que son usadas en la organizaciones de la memoria de instrucciones.

- El Capítulo 3 ha propuesto una herramienta de alto nivel de estimación del consumo energético, que encuentra la configuración óptima de un sistema empotrado para una aplicación y compilador dados, mediante la exploración de diferentes configuraciones de la organización de la memoria de instrucciones.

- El Capítulo 4 ha presentado un análisis de alto nivel de los diferentes compromisos existentes en los esquemas de la arquitectura de *loop buffer* para sistemas empotrados, el cual sirve como directriz a los diseñadores de sistemas empotrados para implementar una organización de la memoria de instrucciones con un bajo coste de energía por tarea realizada.

- El Capítulo 5 ha mostrado como el concepto de *loop buffer* es usado en aplicaciones empotradas reales, que son ampliamente utilizadas en los nodos de sensores inalámbricos biomédicos, con el fin de mostrar qué esquema de *loop buffer* es más adecuado para aplicaciones con cierto comportamiento.
- El Capítulo 6 ha propuesto y analizado las opciones de implementación complementarias y no solapadas de las distintas particiones del espacio de diseño relacionado con la DLB (*Distributed Loop Buffer Architecture with Incompatible Loop-Nest Organisation*).

En la próxima Sección, se resumen las contribuciones principales de esta tesis doctoral.

A.7.1. Contribuciones Principales

Las principales contribuciones que esta tesis doctoral ha ofrecido a la comunidad investigadora son:

- El estudio metódico de las técnicas de optimización de bajo consumo de energía existentes que se utilizan en la organización de la memoria de instrucciones, resumiendo y comparando sus ventajas, inconvenientes y compromisos.
- El uso de simulaciones *post-layout* para evaluar el impacto energético de las arquitecturas de *loop buffer* en el sistema, en una evaluación experimental basada en un método sistemático para obtener una estimación exacta de la actividad de conmutación y de los parámetros parasitarios.
- El desarrollo de una herramienta de estimación energética de alto nivel que, para una aplicación y compilador dados, permite la exploración no sólo de mejoras arquitecturales y configuraciones del compilador, sino también de transformaciones de código que estén relacionadas con la reducción de energía de la organización de la memoria de instrucciones.
- La evaluación de diferentes esquemas de *loop buffer* para ciertas aplicaciones empotradas, guiando al diseñador de sistemas empotrados a tomar la correcta decisión en los compromisos que existen entre el presupuesto de energía, el rendimiento requerido y el coste de área del sistema empotrado.
- Una implementación de una arquitectura de *loop buffer*, que optimiza tanto el consumo de energía dinámico como el consumo de energía relacionado con las corrientes de fugas de la organización de la memoria de instrucciones.

- La propuesta y el análisis de diversas opciones de implementación complementarias y no solapadas para las distintas particiones del espacio de diseño que está relacionado con las arquitecturas DLB.

Cada uno de los Capítulos que forman esta tesis doctoral aborda alguna de las contribuciones que estan anteriormente expuestas. En términos de publicaciones científicas, esta tesis doctoral ha generado los siguientes artículos en revistas internacionales:

ARJ+b13 Artes, A., R. Fasthuber, J. L. Ayala, P. Raghavan, and F. Catthoor, "Design Space Exploration of Loop Buffer Schemes in Embedded Systems", Special Issue of Journal of Systems Architecture on Design Space Exploration of Embedded Systems: Elsevier Amsterdam, 2013.

ARJ+a13 Artes, A., R. Fasthuber, J. L. Ayala, P. Raghavan, and F. Catthoor, "Design Space Exploration of Distributed Loop Buffer Architectures with Incompatible Loop-Nest Organisations in Embedded Systems", Journal of Signal Processing Systems: Springer New York, 2013.

AJJFa12 Artes, A., J. L. Ayala, J. Huisken, and F. Catthoor, "Survey of Low-Energy Techniques for Instruction Memory Organisations in Embedded Systems", Journal of Signal Processing Systems: Springer New York, 2012.

A.JFb12 Artes, A., J. L. Ayala, and F. Catthoor, "Power Impact of Loop Buffer Schemes for Biomedical Wireless Sensor Nodes", Journal of MDPI Sensors: MDPI AG, 2012.

, esta tesis doctoral ha generado los siguientes artículos en conferencias internacionales:

A.JFa12 Artes, A., J. L. Ayala, and F. Catthoor, "IMOSIM: Exploration Tool for Instruction Memory Organisations based on Accurate Cycle-Level Energy Modelling", IEEE International Conference on Electronics, Circuits, and Systems (ICECS), 2012.

AJV+a11 Artes, A., J. L. Ayala, V. A. Sathanur, J. Huisken, and F. Catthoor, "Run-time Self-tuning Banked Loop Buffer Architecture for Power Optimization of Dynamic Workload Applications", IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SOC), 2011.

A. Aa10 Artes, A., "Power Consumption on Loop Buffer based Instruction Memory Organizations", STW.ICT Conference on Research in Information and Communication Technology, 2010.

AFM+a09 Artes, A., F. Duarte, M. Ashouei, J. Huisken, J. L. Ayala, D. Atienza, and F. Catthoor, "Energy Efficiency using Loop Buffer based Instruction Memory Organization", IEEE International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA), 2009.

, y esta tesis doctoral ha contribuido en los siguientes artículos en revistas y congresos internacionales:

HSN+a13 Kim, H., S. Kim, N. V. Helleputte, A. Artes, M. Konijnenburg, J. Huisken, C. V. Hoof, and R. F. Yazicioglu, "A Configurable and Low-Power Mixed Signal SoC for Portable ECG Monitoring Applications", Journal of IEEE Transactions on Biomedical Circuits and Systems: IEEE Computer Society, 2013.

MMJ+a12 Komalan, M., M. Hartmann, J. I. Gomez, C. Tenllado, A. Artes, and F. Catthoor, "System Level Exploration of Resistive-RAM (ReRAM) based Hybrid Instruction Memory Organization", International Memory Architecture and Organization Workshop (MeAOW), 2012.

HRS+a11 Kim, H., R. Firat, S. Kim, V. N. Helleputte, A. Artes, M. Konijnenburg, J. Huisken, J. Penders, and V. C. Hoof, "A Configurable and Low-Power Mixed Signal SoC for Portable ECG Monitoring Applications", IEEE International Symposium on VLSI Technology and Circuits, 2011.

A.7.2. Direcciones Futuras de Investigación

La investigación que se presenta en esta tesis doctoral ha identificado los factores que impactan fuertemente en el consumo de energía de la organización de la memoria la instrucciones de los sistemas empotrados. Además, esta tesis doctoral ha aportado algunas ideas para gestionar de manera eficiente el consumo de energía de este componente del sistema empotrado, guiando a los diseñadores de sistemas empotrados a tomar la decisión correcta en los compromisos que existen entre el presupuesto de energía, el rendimiento requerido y el coste de área del sistema empotrado. Sin embargo, algunos temas interesantes para futura investigación han surgido durante la evolución de este trabajo.

A continuación se proponen las direcciones futuras de investigación y las posibles mejoras del trabajo presentado en esta disertación:

- La herramienta de alto nivel propuesta en el Capítulo 3, que sirve para la estimación de energía y la exploración del espacio de diseño de la organización de la memoria de instrucciones, puede ser ampliada con el propósito de construir un simulador que incorpore no sólo las mejoras de

la organización de la memoria de instrucciones, sino también las mejoras de otros componentes del sistema empotrado.

- Esta tesis doctoral ha propuesto arquitecturas de *loop buffer* que son gestionadas a través de una lógica de control implementada en hardware. Por lo tanto, sería interesante analizar los beneficios de la gestión mediante software de las memorias internas de la arquitectura de *loop buffer* en comparación con la gestión mediante hardware. En este escenario, el compilador sería el responsable de mapear las partes adecuadas de la aplicación en las memorias internas de la arquitectura de *loop buffer*, activando estas memorias sólo cuando se necesiten.

- De la misma manera que una arquitectura CELB ha sido implementada en una aplicación portátil de bajo consumo de potencia para la monitorización de señales ECG [KYS+11], el resto de las arquitecturas de *loop buffer* que han sido propuestas y analizadas en esta tesis doctoral deberían ser fabricadas e introducidas en una implementación real de un sistema empotrado. Esto permitirá corroborar los resultados experimentales que se han obtenido a partir de las simulaciones *post-layout* que se han presentado en esta tesis doctoral.

- Esta tesis doctoral se ha centrado en el estudio, el análisis, la propuesta, la implementación y la evaluación de técnicas de optimización de bajo consumo de energía que pueden ser utilizadas en las organizaciones de la memoria de instrucciones de los sistemas empotrados. Por lo tanto, sería interesante ver el comportamiento de estas mismas técnicas, pero en este caso, aplicándose en otro tipo de sistemas (por ejemplo, ordenadores de escritorio, sistemas de servidores, etc).

Appendix B

Architectural Exploration in the Design of Application-Specific Processors

"You can design and create, and build the most wonderful place in the world. But it takes people to make the dream a reality."

— Walter Elias Disney (Walt Disney).

This Appendix presents the retargetable tool-suite from *Target Compiler Technologies* that has been used to design the processor architectures that are presented throughout this Ph.D. thesis.

B.1. Introduction

Nowadays, the semiconductor industry is driven by the rapidly growing market of smart consumer devices. These products are characterised for been feature-rich, multi-sensing, wirelessly connected, battery powered, and green. Therefore, the design of these smart consumer devices needs to become software programmable in order to cope with the flexibility requirements that the next-generation of these smart consumer devices will required. This technology trend calls for an increased use of ASIP (*Application-Specific Instruction-Set Processor*) designs, and turns SoC (*System-on-Chip*) into heterogeneous multi-core platforms offering significant amounts of multi-threaded parallelism. Figure B.1 shows ASIP designs in heterogeneous multi-core SoCs. These designs are a key technology, because ASIPs not only bring important benefits, but also reconcile what are often considered contradictory requirements: performance, power consumption, and programmability.

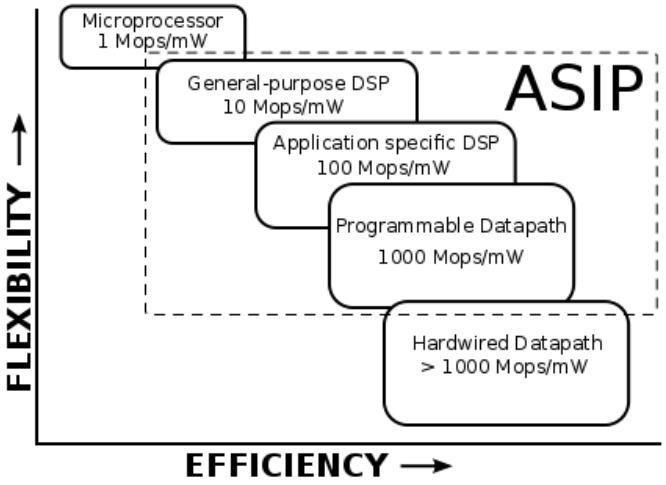


Figure B.1: ASIPs in heterogeneous multi-core SoCs [TAR12].

B.1.1. Performance

ASIP designs boost performance by combining multiple forms of parallelism with specialisation of the architecture. ILP (*Instruction-Level Parallelism*), implemented sometimes as VLIW (*Very Long Instruction Word*) architectures, is a key requirement in order to meet the performance requirements in almost every design. DLP (*Data-Level Parallelism*), implemented sometimes as SIMD (*Single-Instruction Multiple-Data*) architectures or vector processing units, exploits regularities in the application that require identical instructions to operate on multiple data components. TLP (*Thread-Level Parallelism*) is obtained by allocating multiple cores, each one specialised for its tasks at hand. The above forms of parallelism are combined with a specialisation of the architectural resources, in which arithmetic and logic units are included. The structure of the register file, the components of the memories, and the related addressing operators have to be customised to support the data bandwidth required by the parallel architecture of the ASIP design.

B.1.2. Power Consumption

The architectural tricks for performance optimisation discussed above also contribute to reduce the energy consumption. A key point is that, by virtue of increased parallelism and architectural specialisation, the same task can be completed in fewer instruction cycles. Power gating can be used at system level to reduce the leakage consumption. In a heterogeneous multi-core system, depending on the scenario of use not all tasks may be active, in which case certain ASIPs can be powered down for a longer time.

B.1.3. Programmability

The programmability of an ASIP is only effective within its application domain. A well-designed ASIP provides sufficient programmability to cope with late algorithmic changes and bug fixes, to add new features for product differentiation, to ship first while the standard is evolving, or even to extend products to new markets without requiring a silicon re-spin.

B.2. No Efficient ASIP Design without Tools

The key to develop an efficient ASIP design is architectural exploration. Some IP (*Intellectual Property*) vendors offer configurable ASIP solutions that enable exploration within the boundaries of a parametrised, yet confined architectural template. Vendors of retargetable ASIP design tools like *Target Compiler Technologies* [TAR12] take a fundamentally different approach. Their tools read the formal model of an embedded instruction-set processor platform which is expressed in a processor description language (*i.e.*, nML language). nML is the first commercially available high-level definition language, that quickly evolved to become the de-facto standard to describe a processor architecture and an ISA (*Instruction Set Architecture*). The architectural scope of these tools extends beyond parametrised templates, and thus enables true architectural exploration.

Figure B.2 pictures *IP Designer*, the retargetable ASIP design tool from *Target Compiler Technologies* [TAR12] based on the nML processor description language. From a nML model, the tool automatically builds a complete SDK (*Software Development Kit*), including an optimising C compiler, an instruction-set simulator, and an on-chip debugger. The simulator generates extensive profiling reports about instructions, storage, and other hardware resources, indicating the architectural hot-spots to the embedded systems designer. The instantaneous availability of a production-level C compiler for any architecture described in nML is a unique feature of *IP Designer*. Furthermore, *IP Designer* generates a power-optimised RTL (*Register-Transfer Level*) implementation of each ASIP, suited for logic synthesis with all standard third-party synthesis tools.

Architectural exploration includes C compilation, profiling, tuning of the nML model, RTL generation, and logic synthesis. Due to this fact, based on a formal processor description language, these tools are not restricted to parametrised template architectures, an thus enable rapid and true architectural exploration.

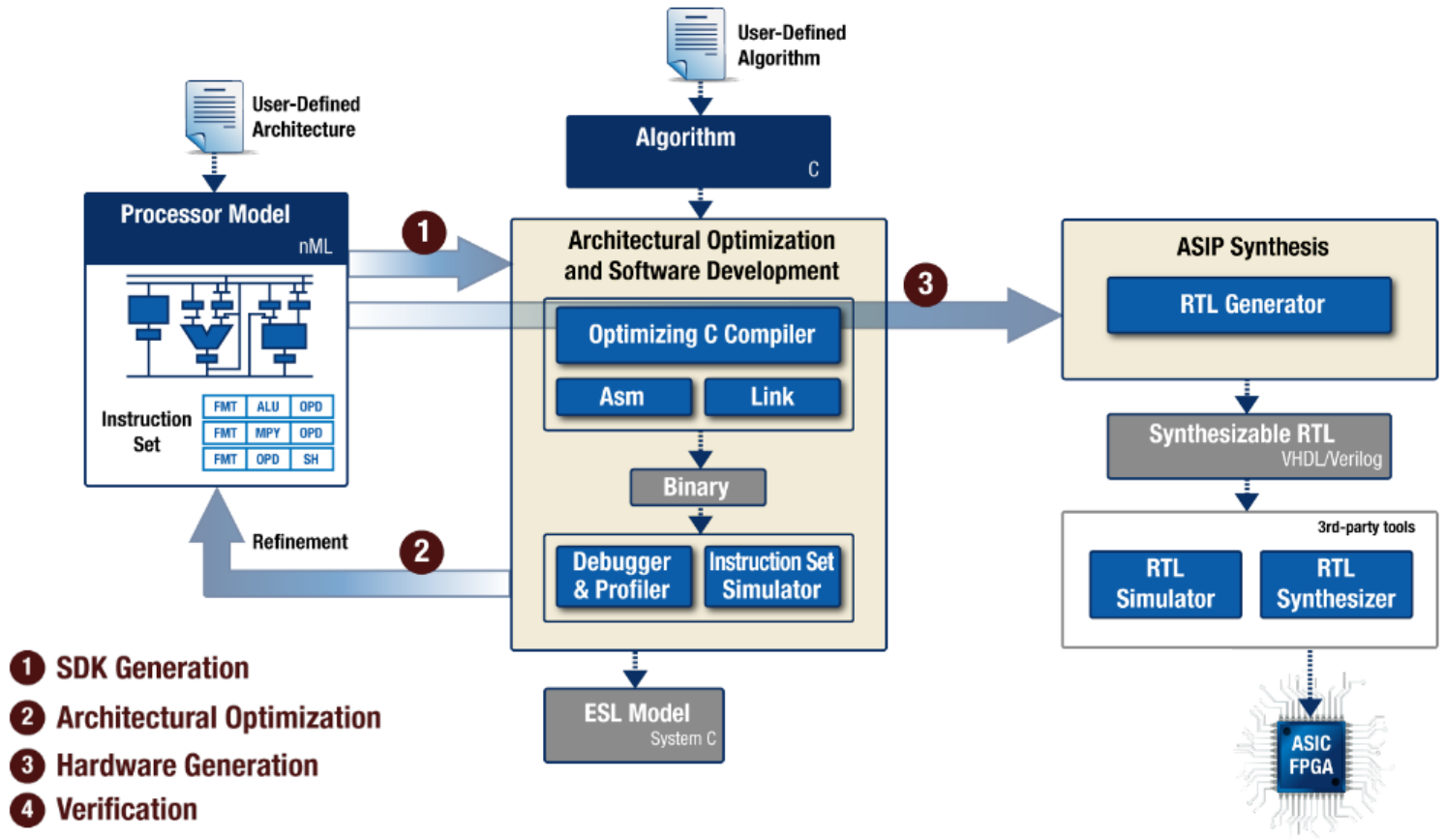


Figure B.2: *IP Designer* tool-suite from *Target Compiler Technologies* [TAR12].

B.3. Template Processor

The general-purpose processor architecture that is used as template in the tools of *Target Compiler Technologies* [TAR12] is presented in Figure B.3. The ISA of this processor architecture is composed of integer arithmetic, bitwise logical, compare, shift, control, and indirect addressing I/O instructions. Apart from support for interrupts and on-chip debugging, this processor architecture supports zero-overhead looping control hardware, which allows fast looping over a block of instructions. Once the loop is set using a special instruction, additional instructions are not needed in order to control the loop, because the loop is executed a pre-specified number of iterations (known at compile time). This loop buffer implementation supports branches, and in cases where the compiler cannot derive the loop count, it is possible to inform the compiler through source code annotations that the corresponding loop will be executed at least N times, and at most M times, such that no initial test is needed in order to check whether the loop has to be skipped. The special instruction that controls the loops introduces only one cycle delay. The status of this dedicated hardware is stored in the following set of special registers:

LS **Loop Start address register.** This register stores the address of the first instruction of the loop.

LE **Loop End address register.** This register stores the address of the last instruction of the loop.

LC **Loop Count register.** This register stores the remaining number of iterations of the loop.

LF **Loop Flag register.** This register keeps track of the hardware loop activity. Its value represents the number of nested loops that are active.

The experimental framework uses an I/O interface in order to provide the capability of receiving and sending data in real-time. This interface is implemented in the processor architecture by FIFO (*First In, First Out*) architectures that are directly connected to the register file. The DM (*Data Memory*) that is required by this processor architecture in order to be a general-purpose processor is a memory with a capacity of 16K words/16 bits, whereas the required PM (*Program Memory*) is a memory with a capacity of 2K words/16 bits.

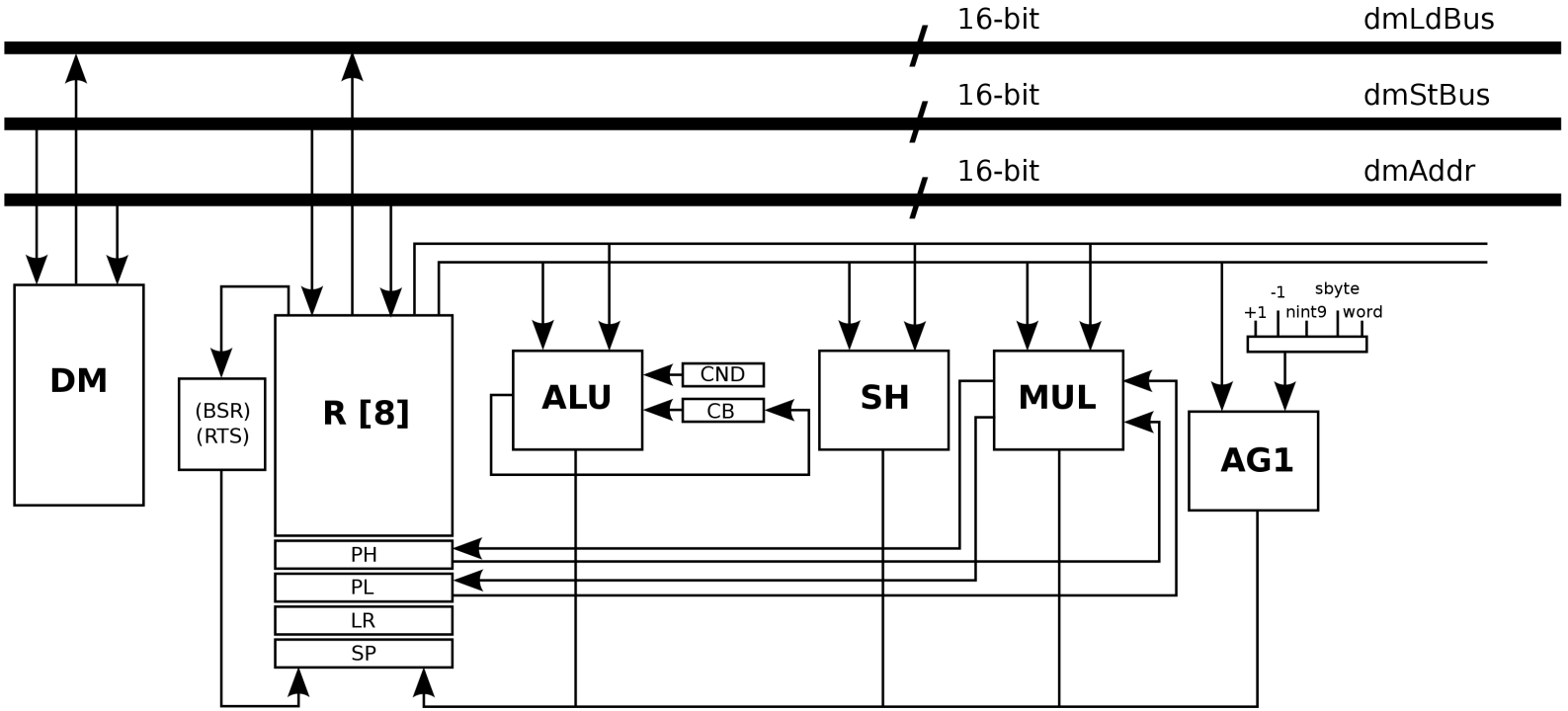


Figure B.3: Data-path of the general-purpose processor used as template processor.

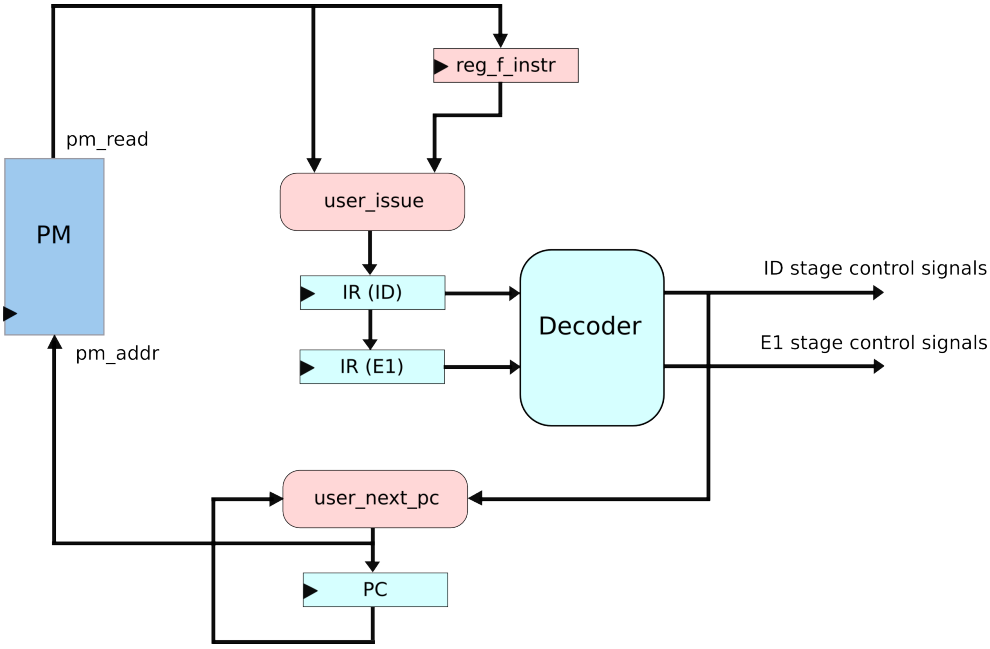


Figure B.4: Control-path of the general-purpose processor used as template processor.

Figure B.3 presents the data-path of this processor architecture, where the main blocks are DM (*Data Memory*), R (*Register File*), ALU (*Arithmetic Logic Unit*), SH (*Shift Unit*), MUL (*Multiplication Unit*), and AG (*Address Generation Unit*). The address generation unit specifies the next address as a normal instruction word in the case of the *word* label, as a negative offset to the stack pointer register in the case of the *nint9* label, and as a relative offset of short jump instructions in the case of the *sbyte* label. In Figure B.4, the main blocks are PM (*Program Memory*), PC (Program Counter), and the registers IR (ID) and IR (E1) which are related to the decode and the execute stage of the processor pipeline.

Appendix C

Benchmarks

"There are no such things as applied sciences, only applications of science."

— Louis Pasteur.

This Appendix presents and describes the real-life embedded applications that are used as benchmarks in this Ph.D. thesis.

C.1. Introduction

Due to the fact that embedded electronic devices are going to be a part of our future daily life, this Ph.D. thesis uses real-life embedded applications as benchmarks to show, analyze, and corroborate the benefits and the disadvantages of each one of the concepts in which this Ph.D. thesis is based on. The selected benchmarks, which are described in this Appendix, are prime examples not only of all application domains that are loop dominated, exhibit sufficient opportunity for DLP (*Data-Level Parallelism*) and/or ILP (*Instruction-Level Parallelism*), comprise signals with multiple word-lengths, and require a relatively limited number of variable multiplications, but also of more general-purpose applications domains that can be found in the area of wireless base-band signal processing, multimedia signal processing, or different types of sensor signal processing. This selection of the benchmarks was performed with the intention of making the analyses that are presented in this Ph.D. thesis generic enough, in order to be applicable to all loop-dominated application domains of the embedded systems.

C.2. Advanced Encryption Stardard Algorithm

The WBAN (*Wireless Body Area Network*) is a WSN (*Wireless Sensor Network*) that is used for communication among sensor nodes operating on, in, or around the human body in order to monitor vital body parameters and movements. A WSN not only has limited resources such as computation capability and memory storage, but also is vulnerable to many kinds of attacks. Security in a WBAN is very important to guarantee and protect the patient's personal sensitive data. Due to this fact, the wireless communication in the WSN mandates the integration of security functionality. The AES (*Advanced Encryption Standard*) [NIS12] is adopted in many WSN standards such as *IEEE 802.15.4* and *IEEE 802.15.6* (see reference [IEE12]). The AES algorithm provides data encryption, but combined with the appropriate functionality, it provides additional security services as well, such as data authentication, data integrity, and replay protection. The AES functionality is not supported by a typical microprocessor, and the hardware support for the AES functionality is mandatory if a high-performance low-energy implementation is required. The reason for this, is that the AES algorithm is executed very frequently in the WSN. Specifically, for each packet that a sensor node is transmitting or receiving, the AES algorithm is executed twice, for encryption/decryption and authentication.

C.2.1. Description of the Algorithm

The AES algorithm is intended for cryptographic applications. The AES algorithm used in this Ph.D. thesis is based on the security operation mode *AES-CCM-32*. This mode of operation provides confidentiality, data integrity, data authentication, and replay protection. In the next paragraphs, AES and CCM are explained in detail.

AES [NIS12] is a FIPS (*Federal Information Processing Standard*) based on a symmetric-key encryption standard, in which both the sender and the receiver use a single key for encryption and decryption. The data block length that is used by this algorithm is fixed to 128 bits, while the length of the cipher key can be 128, 192, or 256 bits, which are represented by 4, 6, or 8 words respectively. Besides, the AES algorithm is an iterative algorithm in which the iterations are called *rounds*, and the total number of rounds can be 10, 12, or 14, depending on whether the key length is 128, 192, or 256 bits, respectively. The 128-bit data block, that is processed during the *rounds*, is divided into 16 bytes that are mapped to a 4 x 4 array called *State*. Every internal operation of the AES algorithm is performed on the *State*. Each byte in the *State* is considered as an element of Galois Fields $GF(2^8)$. The irreducible polynomial used in the AES algorithm to construct the $GF(2^8)$

field is $p(x) = x^8 + x^4 + x^3 + x + 1$. In the encryption process, each *round*, except for the final *round*, consists of the following four transformations:

- **SubBytes.** This transformation is a non-linear byte substitution in which each byte independently is replaced by another. This transformation can be implemented in software in two ways: based on finite fields digital logic or as a look-up table (S-Box lut).
- **ShiftRows.** This transformation is a transposition step in which the bytes of the last three rows of the *State* are shifted cyclically based on different offsets. This transformation has the effect of moving bytes to lower positions in the row, while the lowest bytes wrap around into the top of the row.
- **MixColumns.** This transformation is a mixing operation which operates on the columns of the *State*, combining the four bytes of each column using a linear transformation. Specifically, the columns are considered as polynomials over $GF(2^8)$ and are multiplied modulo x^4+1 with a fixed polynomial $a(x)$, given by $a(x) = 3 \times x^3 + 1 \times x^2 + 1 \times x + 2$.
- **AddRoundKey.** This transformation applies a bitwise XOR operation between the *State* and a Round key. Each round key has a size of 4 words and is derived from the cipher key using a key schedule.

The final *round* does not have the MixColumns transformation. Figure C.1 shows the flowchart of this algorithm when it is working in encryption mode.

The CCM (CTR-CBC-MAC), which is presented in the NIST (*National Institute of Standards and Technology*) Special Publication *800-38C* [Dwo04], encrypts and authenticates the message and the associated data. Depending on the size of the message authentication code that it produces (4, 8, or 16 bytes), three different variations of *AES-CCM* exist: *AES-CCM-32*, *AES-CCM-64*, and *AES-CCM-128*.

Due to the fact that WBANs have ultra-low power requirements, the proposed algorithm supports only 128-bit key. In addition, only the encryption mode of the AES algorithm is supported. However, with a very small change in the design, both encryption and decryption can be supported. In this algorithm, the input data frame is fixed to 1, 460 bytes of information, whereas the output is a data packet where the information is encrypted.

In this Ph.D. thesis, this algorithm is executed on the general-purpose processor architecture that is described in Section B.3.

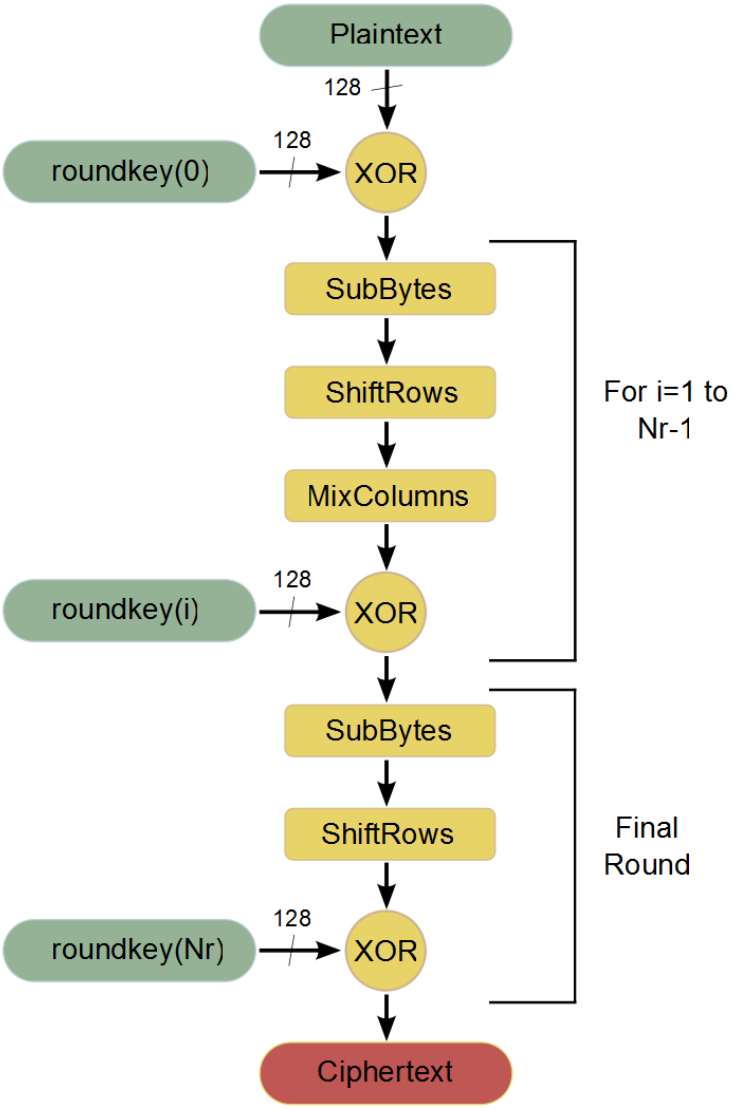


Figure C.1: Flowchart of the AES algorithm. Encryption process.

C.2.2. Profiling Information

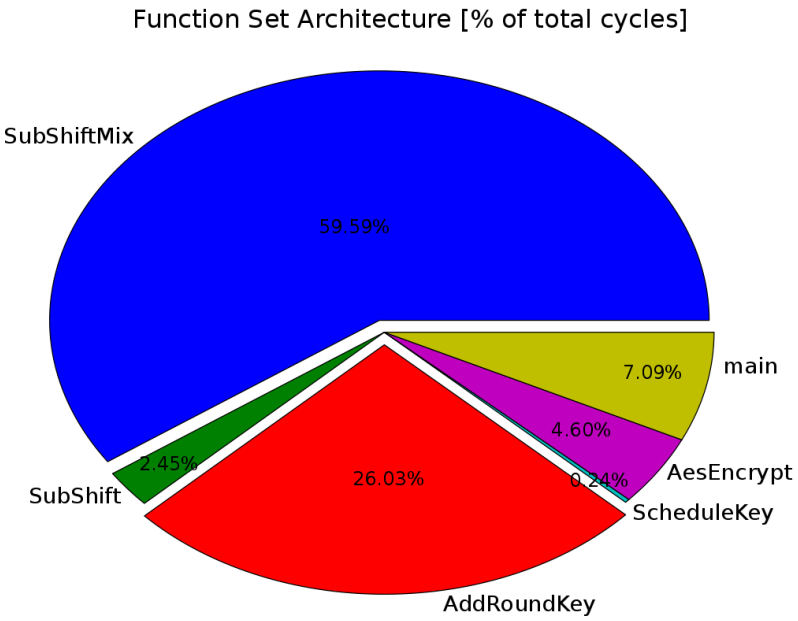


Figure C.2: Function set architecture of the AES algorithm.

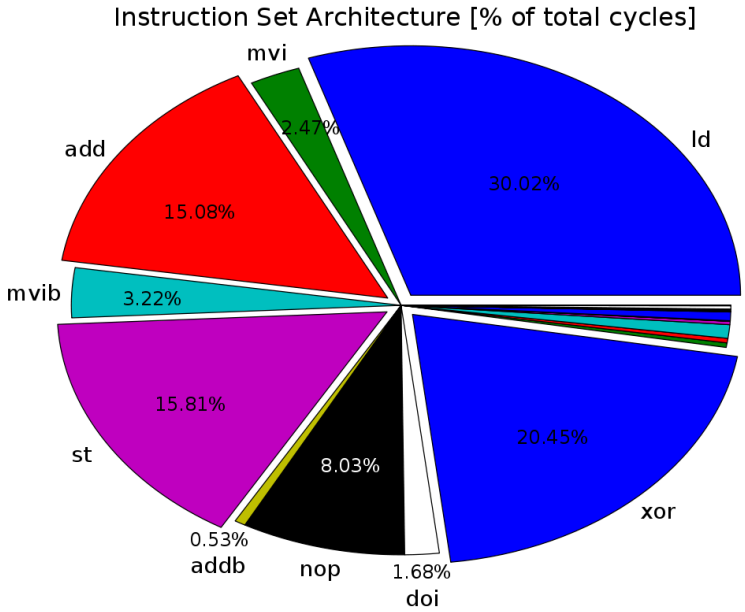


Figure C.3: Instruction set architecture of the AES algorithm.

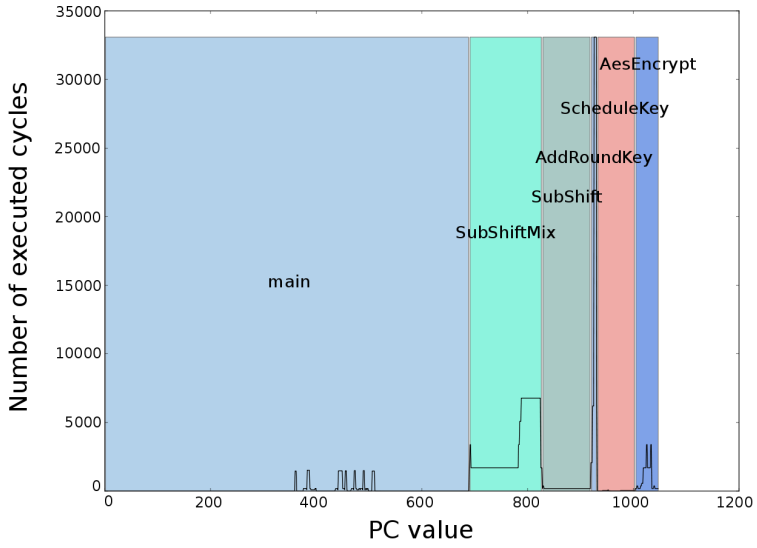


Figure C.4: Program memory footprint of the AES algorithm.

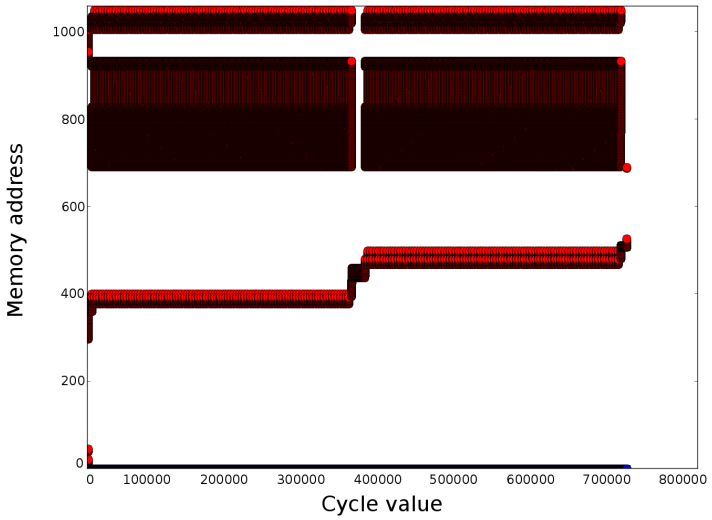


Figure C.5: Profiling information about the access history in the PM of the AES algorithm.

C.3. Advanced Encryption Stardard Algorithm - Optimised Version

The optimised version of the AES (*Advanced Encryption Standard*) algorithm is based on the algorithm presented in Section C.2. This Section presents the modifications and the optimisations that are performed not only to implement the optimised version of the AES (*Advanced Encryption Standard*) algorithm, but also to build the optimised processor architecture over which this optimised algorith is running.

C.3.1. Description of the Algorithm

Analysing the algorithm that is presented and described in Section C.2, the critical functions were identified and optimised in order to improve performance in terms of clock cycles and memory accesses. Custom techniques like source code transformations (*e.g.*, function combination, loop unrolling) and mapping optimisations (*e.g.*, use of look-up tables, elimination of divisions and multiplications, instruction set extensions) were applied to have as outcome a more efficient code.

The following source code transformations were applied:

- **Function Combination.** During a function call, the present *state* (*e.g.*, returning address, registers) has to be saved in the data memory and then retrieved at the end of the execution. Therefore, with the combination of multiple functions in one, it is possible to decrease the memory accesses. The combined functions are not data dependent, and therefore, it is possible to merge them.

- **Loop Unrolling.** Instead of using a loop construct and an iterator index to perform the same operations at different sets of data, the context of the loop can be repeated multiple times. Although this technique leads to an increase in the code size, it usually also improves the performance, because it eliminates the calculation of the array indices based on the loop counter. The loop unrolling was efficiently applied at the 9 *rounds* of AES, leading to a reduction in clock cycles.

Also, mapping optimisation techniques were applied:

- **Use of look-up tables.** Two of the AES functions are based on Galois Fields, and therefore, they can be implemented based on two approaches: with mathematics and with look-up tables. The first approach is computationally demanding which means that it needs many execution

cycles and accesses in memory, while the second approach can be demanding in memory area but it is much faster and perform less memory accesses compared to the first approach. As this design was focused on energy consumption, the look-up table approach was selected.

- **Elimination of divisions and multiplications.** It is always preferable to substitute computationally demanding operations such as multiplications and divisions by lower overhead instructions. In this application, every division and multiplication has divisors or multipliers, which are powers of two. Therefore, these operations are replaced with right or left shift operations accordingly. Specifically, a division or a multiplication with 2^n, is equivalent to a right or a left shift by n bits accordingly.

- **Instruction set extensions.** This technique, which is the main advantage of ASIP (*Application-Specific Instruction-Set Processor*) designs, extends the instruction set of a processor with customised operations for the specific application. The result is the improvement of the performance of the application mapped on the processor architecture.

In the design of this optimised processor, the structure of the general-purpose architecture is kept intact (16-bit data-path), and an extra 128-bit data-path is added. This last data-path is connected with a vector memory, a vector register file, and a vector unit. The vector unit includes the AES accelerating operations, as well as the logic and the arithmetic instructions that this algorithm requires. In this processor, the ISA (*Instruction Set Architecture*) was also extended with one AES accelerating instruction that has two inputs: a 128-bit input which can be the *State* or a *Round* key, and an integer input which indicates the behaviour of the instruction itself. Accordingly to the input, the output contains the *State* or a *Round* key. One of the advantages of this design is the ability to use the larger vector units only when they are required.

All the optimisations and modifications that are presented in this Section result in the new processor architecture that is shown in Figure C.6. Basically, an extra 128-bit data-path is added. This extra data-path includes a Vector Memory (VM), a Vector register file (V), and a Vector Unit (Functional Vector Unit). In order to handle an input signal of 1,460 bytes, the data memory required by this processor architecture is a memory with a capacity of 1k words/16 bits, and the VM is a memory with a capacity of 64 words/128 bits. On the other hand, the required program memory is a memory with a capacity of 1k words/16 bits. This optimised processor is an implementation that is based on the work presented in Reference [TSH+10].

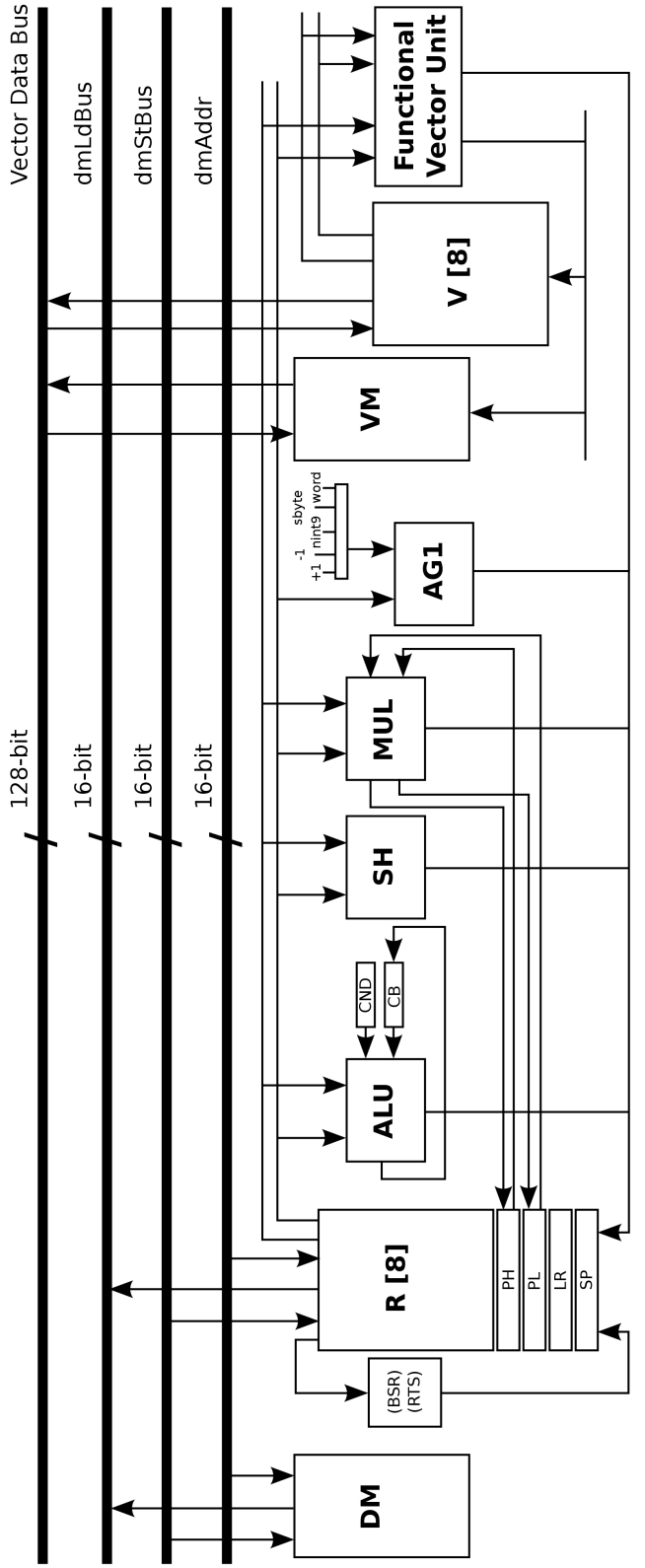


Figure C.6: Data-path of the processor architecture that is optimised for the AES algorithm.

C.3.2. Profiling Information

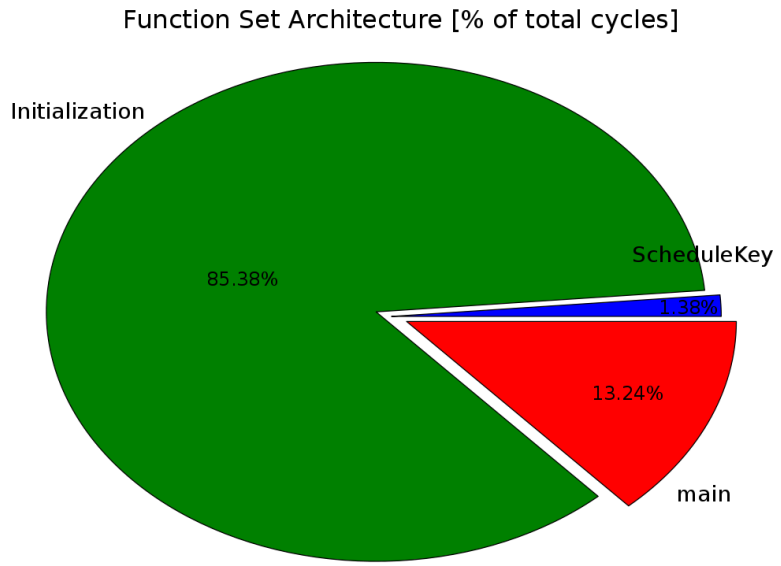


Figure C.7: Function set architecture of the optimised version of the AES algorithm.

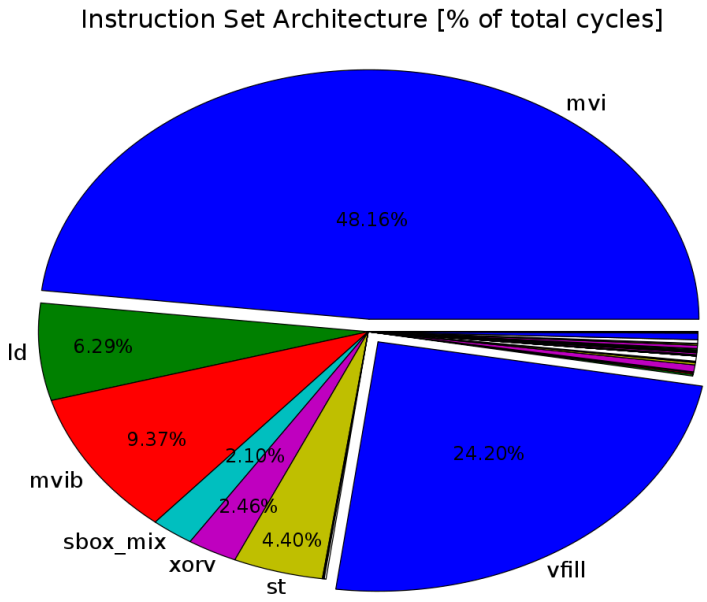


Figure C.8: Instruction set architecture of the optimised version of the AES algorithm.

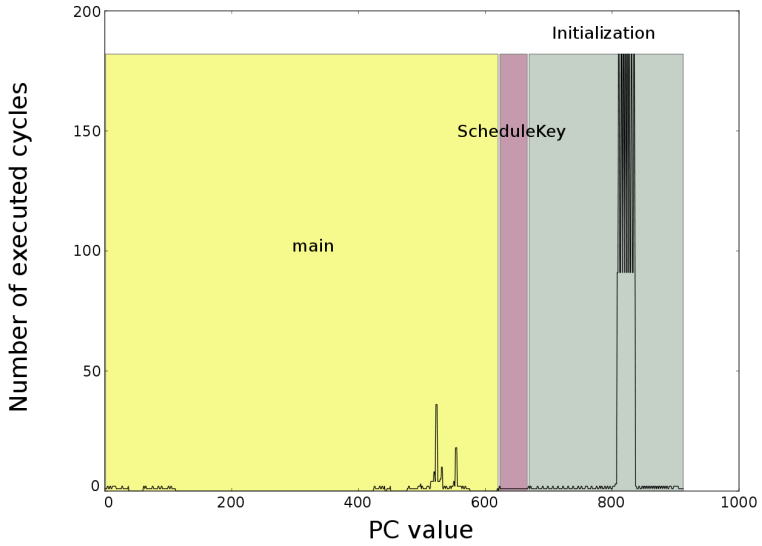


Figure C.9: Program memory footprint of the optimised version of the AES algorithm.

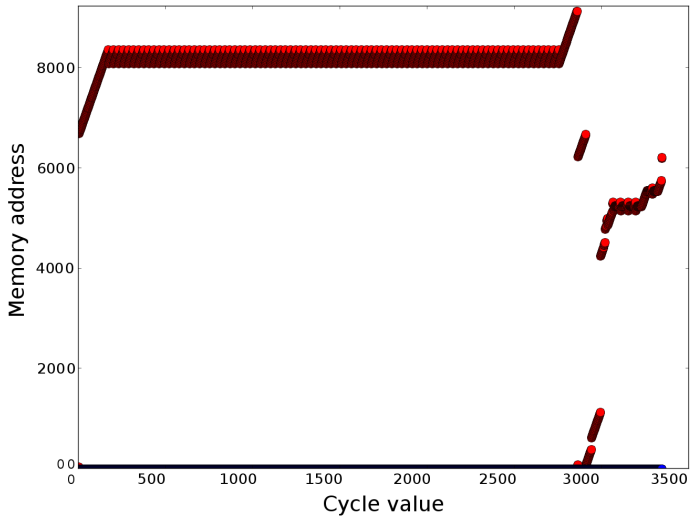


Figure C.10: Profiling information about the access history in the PM of the optimised version of the AES algorithm.

C.4. Bio-imaging Algorithm

This application is a bio-imaging approach proposed by the *BIORES* group at the *K. U. Leuven* [LVB+06]. This approach is an on-line monitoring application that is capable of collecting information about the environment that is observed by the system through a computer vision application. The application uses the analysis of these data to specify many aspects of the monitored object, such as its position, its movement, or even its behavior. The system consists of low-cost intelligent sensors that are combined with image analysis techniques to provide an automated objective contact-less monitoring method for the behavior of the living organisms. The proposed application is based on two modules. The first module detects the monitored object and the second module tracks it. The monitored object is detected by a detection algorithm using image processing techniques on a frame captured by a video camera. Then, a tracking algorithm is executed in order to locate, each time, the position and the characteristics of the monitored object that have changed. These changes over its characteristics are categorized, and finally, translated to the behavior of the monitored animal. Figure C.11 provides the flowchart of the selected bio-imaging application.

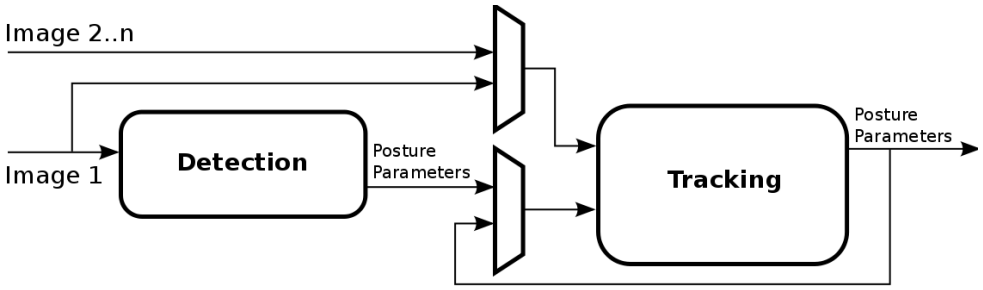


Figure C.11: Flowchart of the Bio-imaging application.

C.4.1. Description of the Algorithm

The detection algorithm exhibits a faster execution time and significantly lower energy consumption than the detection algorithm due to the non-demanding image processing analysis of the tracking algorithm. The detection of the monitored object is performed on the input frame by an image processing algorithm. The algorithm determines the position, the orientation, the body length, and the width of the monitored object through a set of parameters that are based on an ellipse shape mode (called ellipse parameters). After the calculation of these parameters, these are translated to posture parameters, which are the final output of the detection algorithm.

The flowchart of the detection algorithm of the bio-imaging application is illustrated in Figure C.12. The fact that makes the detection algorithm the main bottleneck of this application is the Gaussian Filter that is applied in order to reduce the existing noise of the images that are analyzed. The result of the application of the Gaussian Filter in a whole image is its blurring in order to create a less noisy picture.

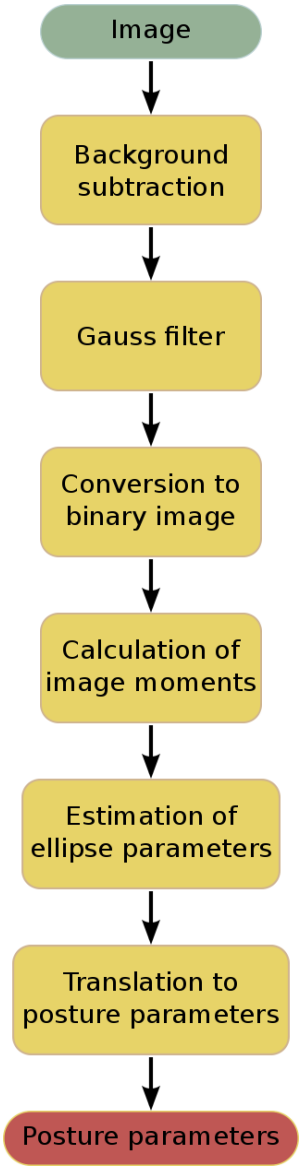


Figure C.12: Flowchart of the detection algorithm that composes the Bio-imaging application.

This algorithm uses the concept *Soft-SIMD* which is described in reference [CRL+10]. The effective application of the *Soft-SIMD* approach [Lam09] is applied especially on the critical loop that forms the Gaussian Filter of the detection algorithm, which is a combination of both hardware and software SIMD (*Single-Instruction Multiple-Data*) without the need for any special hardware support. The re-design of the data-parallel data-path, also referred to as *Soft-SIMD* architecture, was necessary in order to achieve the instruction encoding optimization. The approach that is used for the required masking operations related to the *soft-SIMD* exploits the use of a so-called Shuffler. Instead of using always the worst-case subword size, an intermediate subword size can be used. Whenever a size is changed, repacking operations are applied to the already packed words by the Shuffler. The analysis of the operations performed on the critical loop of the Gaussia Filter of the detection algorithm determines the minimal intermediate subword size that should be used for the packing of the data. The proposed architecture for the implementation of *Soft-SIMD* consists of one shifter, one adder, and one Shuffler as shown in Figure C.13.

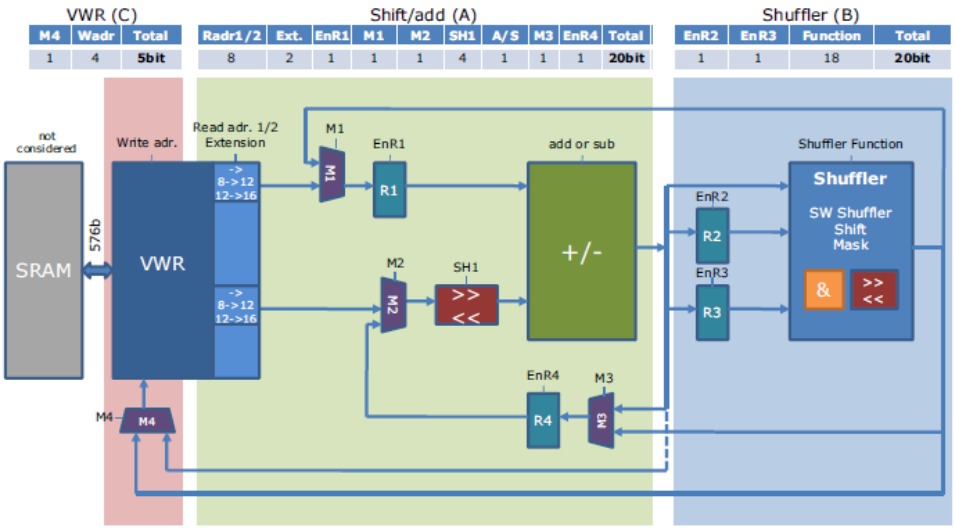


Figure C.13: Instruction Set Architecture of the *Soft-SIMD* processor architecture.

Figure C.14 shows the most important part of the program code of the detection algorithm. As it is possible to see, the program code is formed by two loops in which loop transformations are applied to make the program code as efficent as possible from the energy and performance point of view. Figure C.15 shows how the program code is mapped in the architecture in order to distribute the resourses of the instruction clusters between a variety of possible

tasks, optimizing it in order to place the instructions as efficiently as possible. In Figure C.14 and Figure C.15 the empty spaces are NOP instructions.

```
                    Shift/add (A)          Shuffler (B)      VWR (C)
// LOOP 1                                                    Unrolled loop
for i = 1 ... x:                                                  ↓
        A0: VWR read, +          |                        | C0: VWR write[0]
        A1: VWR read, >>, -      |                        |
        A2: >>, +                | B0: >>                 | C1: VWR write[1]
        A3: VWR read, +          | B1: repack             |
                                 | B2: repack             | C0: VWR write[0]
                                 | B3: repack             |
        A4: >>, -                |                        | C1: VWR write[1]
        A5: VWR read, >>, +      |                        |
end

                                          Unrolled loop
// LOOP 2                                       ↓
for i = 1 ... y:
        A1: VWR read, >>, -      | B0: >>                 | C0: VWR write[0]
        A2: >>, +                | B1: repack             |
        A3: VWR read, +          | B0: >>                 | C1: VWR write[1]
        A4: >>, -                | B1: repack             |
end
```

Figure C.14: Program code of the detection algorithm that composes the Bio-imaging application.

Loop	Sh/Add/VWR read (20b)	Shuffler (20b)	VWR write (5b)
LOOP 1	A0		C0
	A1		
	A2	B0	C1
	A3	B1	
		B2	C0
		B3	
	A4		C1
	A5		
LOOP 2	A1	B0	C0
	A2	B1	
	A3	B0	C1
	A4	B1	

Figure C.15: Scheduling of the detection algorithm that composes the Bio-imaging application.

C.4.2. Profiling Information

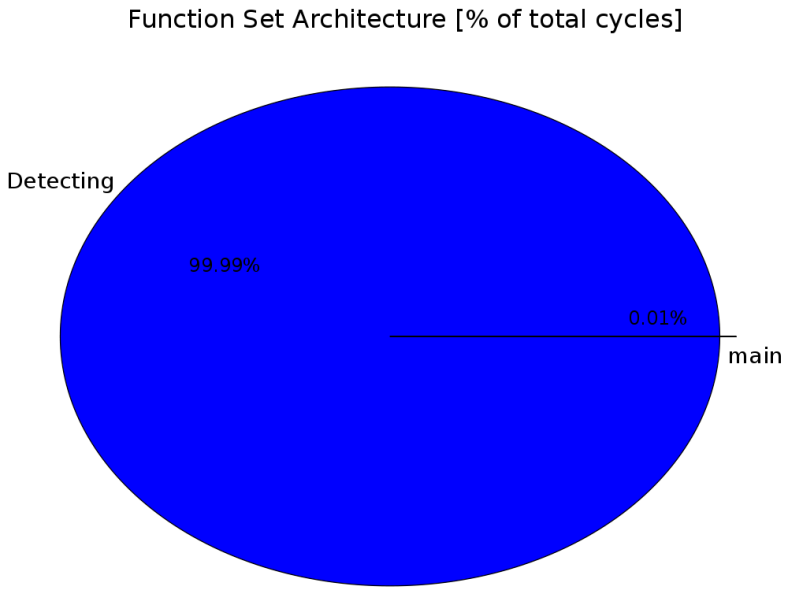


Figure C.16: Function set architecture of the Bio-imaging algorithm.

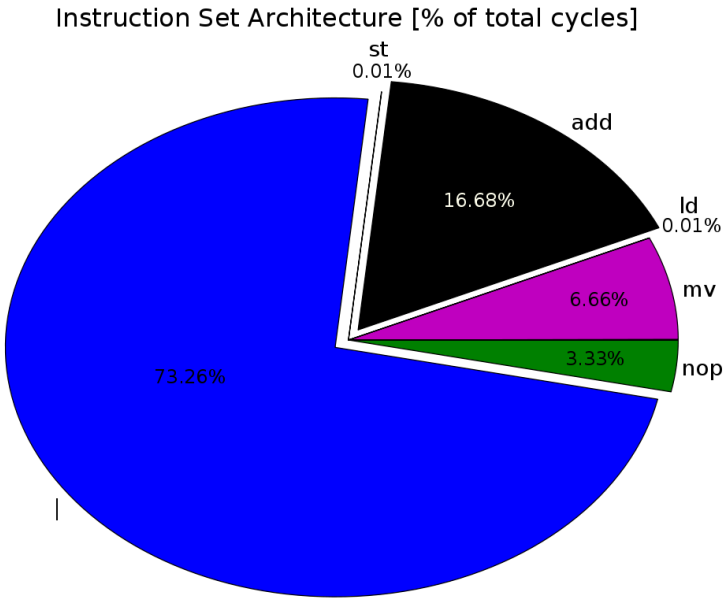


Figure C.17: Instruction set architecture of the Bio-imaging algorithm.

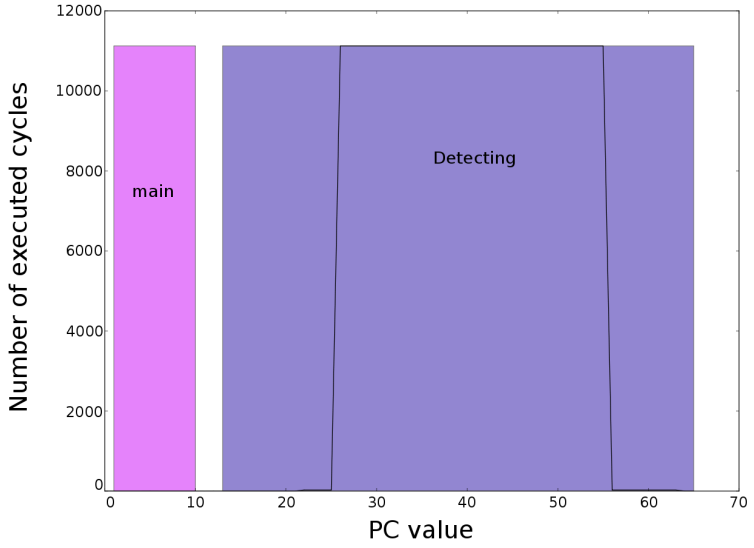


Figure C.18: Program memory footprint of the Bio-imaging algorithm.

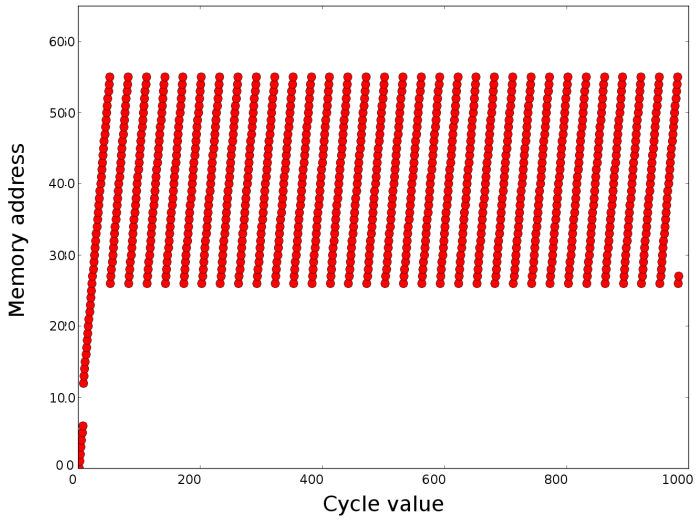


Figure C.19: Profiling information about the access history in the PM of the Bio-imaging algorithm.

C.5. Continuos Wavelet Transform Algorithm

The Wavelet Transform of a continuous time signal, $x(t)$, is definded as:

$$T(a,b) = \frac{1}{\sqrt{a}} \int_{-\infty}^{+\infty} x(t)\psi^*(\frac{t-b}{a})\delta t \tag{C.1}$$

, where $\psi^*(t)$ is the complex conjugate of a wavelet function $\psi(t)$, a is the dilation parameter of the wavelet, and b is the location parameter of the wavelet.

The mathematical criteria that has to be satisfied in order to be classified as a wavelet is:

1. A wavelet must have a finite energy:

$$E = \int_{-\infty}^{+\infty} |\psi(t)|^2 \delta t < \infty \tag{C.2}$$

2. If $\phi(f)$ is the Fourier Transform of the Wavelet Transform $\psi(t)$ (*i.e.*, $\phi(f) = \int_{-\infty}^{+\infty} \psi(t)e^{-jwt}\delta t$), then the *admissibility condition* must be hold:

$$C_g = \int_0^{+\infty} \frac{|\phi(\omega)|^2}{\omega} \delta\omega < \infty \tag{C.3}$$

   , where the Wavelet Transform has no zero frequency component. C_g is called the admissibility constant, and its value depends on the chosen wavelet.

3. For complex or analytic wavelets, the Fourier Transform must be both real and vanish for negative frequencies.

For more details, please check the introduction to the theory and the applications of the Wavelet Transform performed by R. M. Rao *et al.* [RB99].

C.5.1. Description of the Algorithm

The CWT (*Continuous Wavelet Transform*) is one of the building blocks of the biomedical application that is based on a previous algorithm that was developed by I. Romero Legarreta *et al.* [LAR+05]. This algorithm uses the

CWT algorithm [YQ04] to detect heartbeats automatically. The QRS complex is the part of the ECG (*Electrocardiogram*) signal that represents the greatest deflection from the baseline of the signal. In this place is where this algorithm tries to detect the R-peak. Figure C.20 shows the P, Q, R, S, and T waves on an ECG signal.

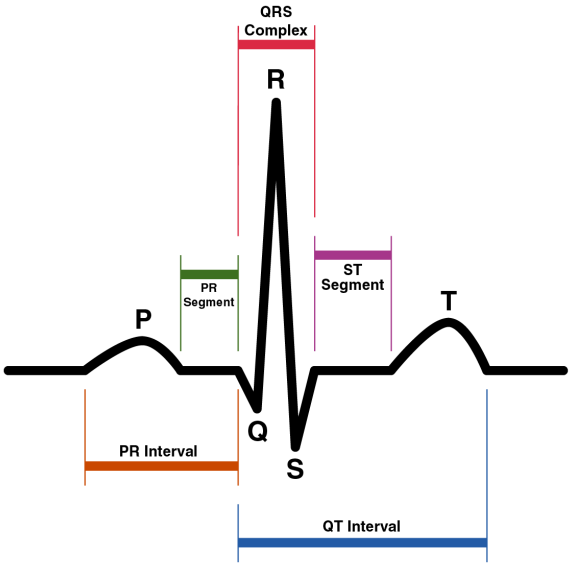


Figure C.20: P, Q, R, S and T waves on an ECG signal.

This algorithm is an optimised C-language version for WBANs. It does not require pre-filtering and it is robust against interfering signals under ambulatory monitoring conditions. The algorithm works with an input frame of 3 seconds, which includes two overlaps of 0.5 seconds between consecutive frames in order to not lose data between frames. Figure C.21 shows the flowchart of this algorithm. The algorithm performs the following steps to process an input data frame:

1. The ECG signal is analysed within a window of 3 seconds, where the CWT algorithm is calculated over this interval and a mask is applied to remove edge components.
2. The square of the modulus maxima of the CWT is taken in order to emphasise the differences between coefficients. Values above a chosen threshold are selected as possible R-peaks.
3. In order to separate the different peaks, all modulus maxima points within intervals of 0.25 seconds are analysed in turn as search intervals. In every search interval, the point with the maximum coefficient value is selected as R-peak.
4. The algorithm finds the exact location of the R-peak in the time-domain.

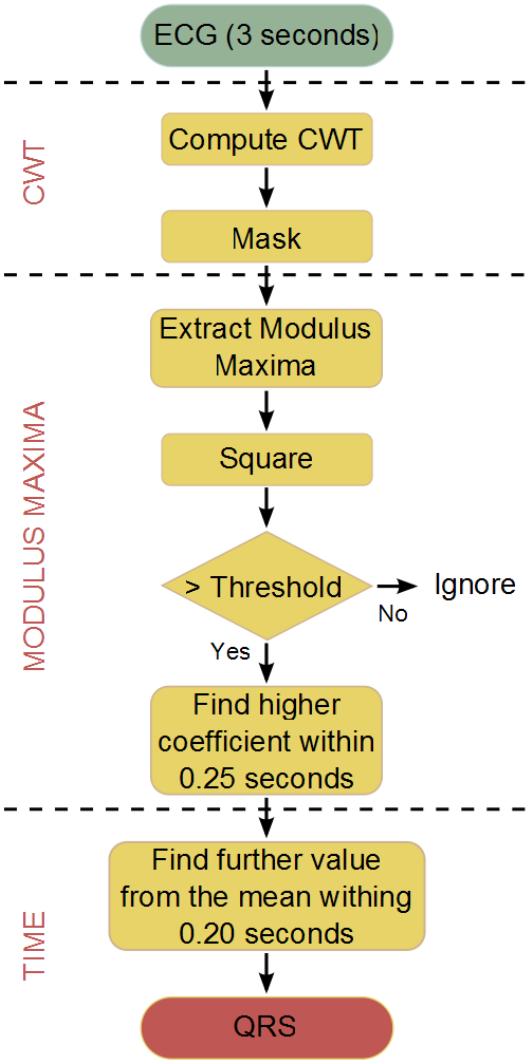


Figure C.21: Flowchart of the heartbeat detection algorithm.

The input of this algorithm is an ECG signal from *MIT/BIH* database [GAG+00]. The output is the positions in time-domain of the heartbeats that are included in the input frame. The testing of this algorithm results in a sensitivity of 99.68 % and a positive predictivity of 99.75 % on the *MIT/BIH* database.

C.5.2. Profiling Information

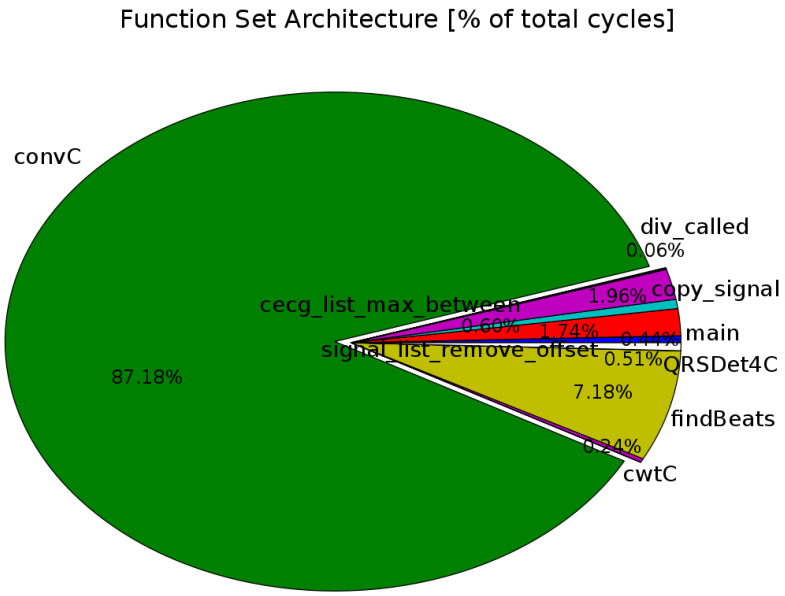


Figure C.22: Function set architecture of the CWT algorithm.

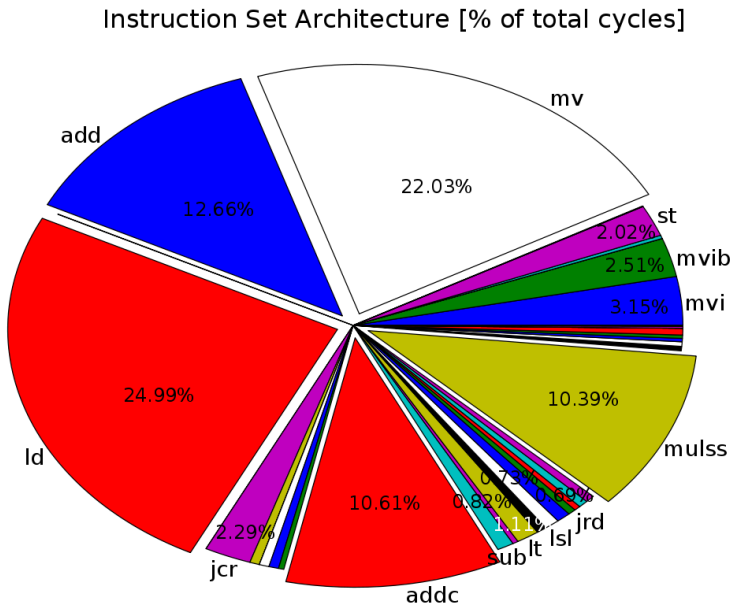


Figure C.23: Instruction set architecture of the CWT algorithm.

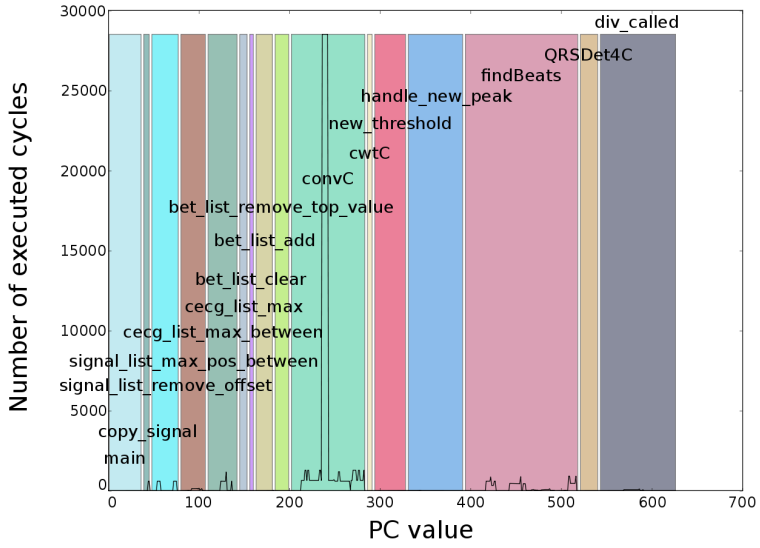


Figure C.24: Program memory footprint of the CWT algorithm.

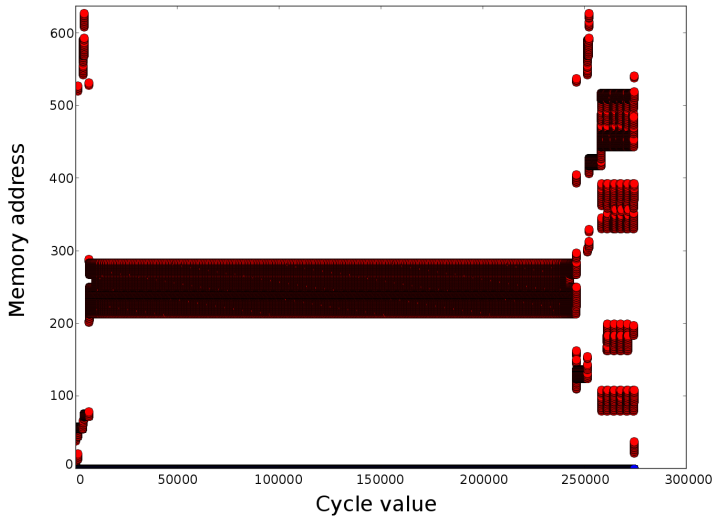


Figure C.25: Profiling information about the access history in the PM of the CWT algorithm.

C.6. Continuos Wavelet Transform Algorithm - Optimised Version

The optimised version of the CWT (*Continuous Wavelet Transform*) algorithm is based on the algorithm presented in Section C.5. This Section presents the modifications and the optimisations that are performed not only to implement the optimised version of the CWT algorithm, but also to build the optimised processor over which this optimised algorithm is running.

C.6.1. Description of the Algorithm

A loop was pointed out as bottleneck of the performance of the algorithm described in Section C.5. This critical loop is contained in the convolution operation within the CWT. A shift operation performed in this loop forces some variables to be defined as *long*. Using a processor with 32-bit data-path instead of 16-bit data-path (*i.e.*, the processor presented in Section B.3), it is possible to decrease the execution time. There is always a trade-off between the complexity of the processor and its energy consumption. However, for this specific scenario, it is a benefit to have a processor of 32-bit data-path. Due to the decision to have a 32-bit data-path processor, an extension of the addressing mode and the word data type of the processor were required. Besides, these modifications required a change in all the instructions contained in the ISA (*Instruction Set Architecture*) that were related to immediate values.

From the deep analysis that has to be performed in order to design the ASIP (*Application-Specific Instruction-Set Processor*) for the CWT algorithm, a loop is pointed out as bottleneck of the performance of in this specific algorithm. This loop performs the convolution operation which is the core of the CWT. A signed multiplication, which result is accumulated in a temporally variable, is performed inside of this critical loop. The execution of this instruction is 72 % of the execution time of the algorithm according to profiling information. Therefore, in order to improve the performance, the MUL unit is modified to multiply two signed integers and accumulate, without shifting, the result of the multiplication. This optimisation saves energy reducing at the same time the complexity of the MUL unit and the execution time of the application.

The load operations that are related to the previous MUL operation are combined in a customised instruction in order to be executed in parallel. However, in the general-purpose processor, it is only possible to load and to store data from the same memory once per stage of the pipeline. In order

to solve this bottleneck, the main data memory is split in two identical data memories: DM (*Data Memory*) and CM (Constant Memory). In order to access two memories in parallel, another address generator AG2 is created such that the load and the store operations from the DM and CM can be performed at the same stage of the pipeline.

As the input registers of the MUL unit can be loaded directly, a new modification can be performed. The parallel load and MUL instruction are combined, by adding another stage in the pipeline and creating a custom instruction that integrates both instructions. The MUL instruction is then executed in the second stage of the pipeline, while the parallel load instruction is executed in the first stage of the pipeline. After this last modification, the MUL operation that is included in the main critical loop of this algorithm is performed using only one assembly instruction.

In a similar way as the MUL operation, another critical loop is optimised by combining load, select, and equal instructions in order to be executed in parallel. This instruction is created adding the functionality of the equal and the select instruction, and combining both of them with a normal load operation. The functional unit ALU 2 is created for this specific operation.

It is necessary to remark that, apart from the specialised instructions that are described in previous paragraphs, custom techniques like source code transformations (*e.g.*, function combination, loop unrolling) and mapping optimisations (*e.g.*, use of look-up tables, elimination of divisions and multiplications, instruction set extensions) are applied to have as outcome a more efficient code.

All the optimisations and the modifications that are described in this Section result in a new processor architecture which is shown in Figure C.26. Basically, an address generator (AG2) and a second ALU (ALU 2) are added, in addition to some pipes and ports. Apart from that, the PC (*Personal Computer*) is modified to handle instruction words that use 32-bit immediate values. In order to handle ECG signals sampled at 1KHz, the memories that are required by this processor architecture are a DM with a capacity of 8K words/32 bits, and a CM with a capacity of 8K words/32 bits. Besides, the program memory that is required by this processor architecture is a memory with a capacity of 1K words/20 bits. This optimised processor is an implementation that is based on the work presented in reference [YKH+09].

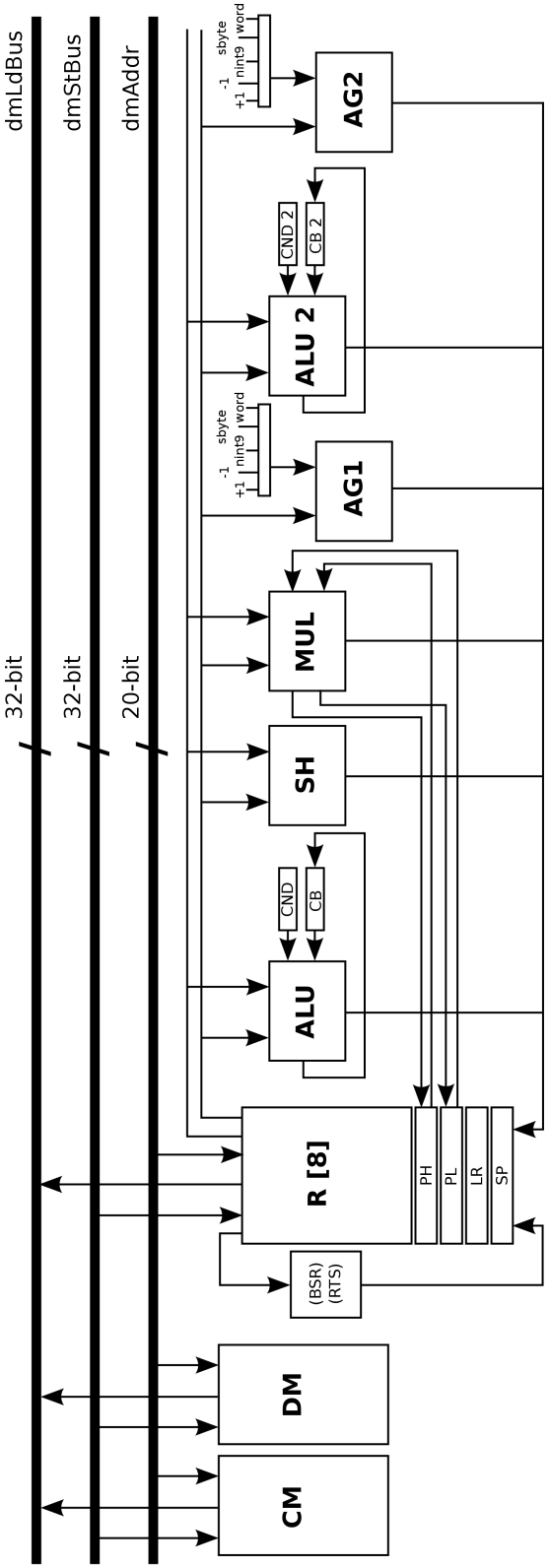


Figure C.26: Data-path of the processor architecture that is optimised for the HDB algorithm.

C.6.2. Profiling Information

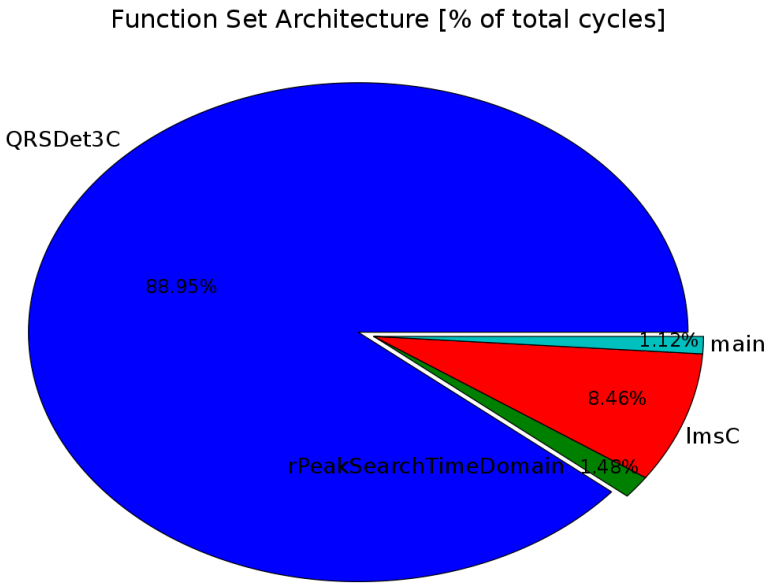


Figure C.27: Function set architecture of the optimised version of the CWT algorithm.

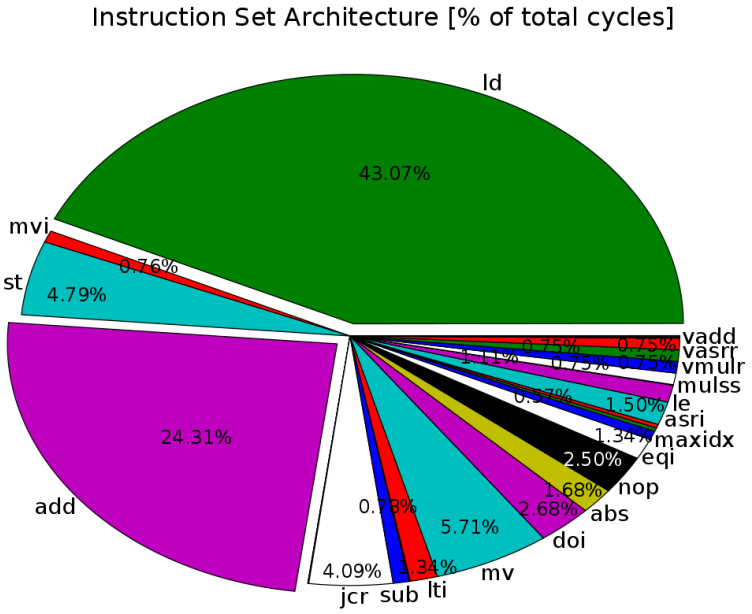


Figure C.28: Instruction set architecture of the optimised version of the CWT algorithm.

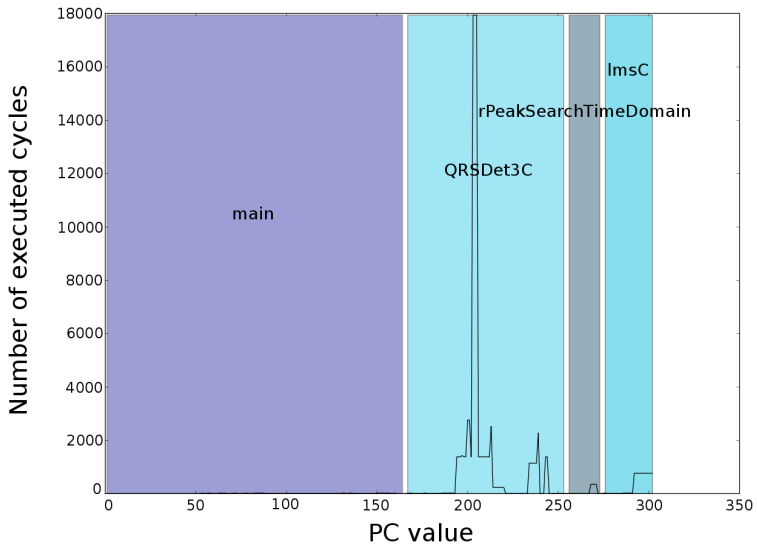


Figure C.29: Program memory footprint of the optimised version of the CWT algorithm.

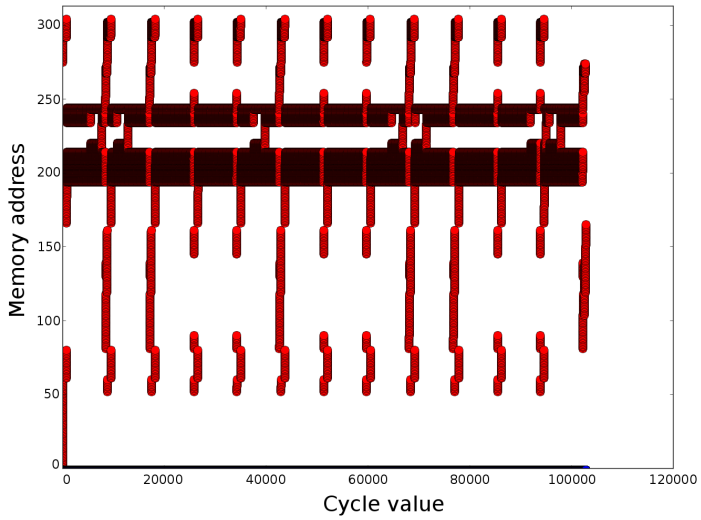


Figure C.30: Profiling information about the access history in the PM of the optimised version of the CWT algorithm.

C.7. Discrete Wavelet Transform Algorithm

The DWT (*Discrete Wavelet Transform*) is used quite often for feature extraction in EEG signals since it can maintain both the time resolution and the frequency resolution of a signal, which is crucial for non-stationary signals such as EEG data. However, the memory usage is large since EEG data is usually recorded over hours and days. Thus, real-time implementation with little amount of memory usage is important to make it feasible on a DSP (*Digital Signal Processor*). This application is a real-time implementation of the daubechies-4 DWT algorithm, in which it is shown that the memory usage is greatly reduced over vector implementation.

C.7.1. Description of the Algorithm

The EEG (*Electroencephalogram*) data are used frequently for health care monitoring and diagnosis, such as epileptic seizure detection, emotion monitoring, sleep monitoring, etc. In case of epileptic seizure detection, one of the early signs of seizure is the presence of characteristic transient waveforms (spikes and sharp waves) in EEG data. Figure C.31 shows the international 10-20 system of the location of the electrodes when measuring the EEG data [LCL+07].

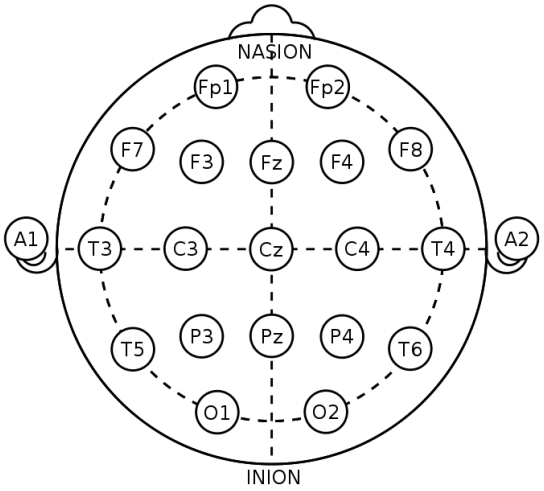


Figure C.31: International system of the location of the electrodes for EEG [LCL+07].

In order to extract useful information such as spikes from the EEG data, a complete algorithmic flow is needed. Figure C.32 depicts the flowchart of the DWT algorithm.

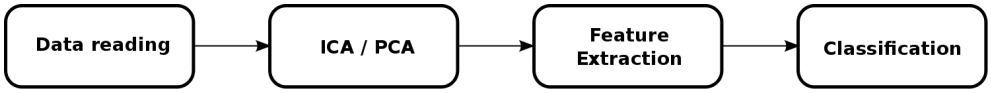


Figure C.32: Flowchart of the DWT algorithm.

In this flow, each box is explained as follows:

1. **Data reading:** read the EEG signal.
2. **ICA/PCA:** EEG data recorded on electrodes or channels are not sensitive enough to pick out individual action potentials. ICA (*Individual Component Analysis*) or PCA (*Principle Component Analysis*) involves a linear change of basis of EEG data to spatially transformed virtual channel basis.
3. **Feature Extraction:** extract salient features from the input data. Wavelet Transform or DWT is often applied here to extract both time and frequency information as EEG data are often non-stationary.
4. **Classification:** to provide decision boundaries based on learning from the features which are assigned with classes.

As it was stated, EEG data are non-stationary, which makes the traditional FFT (*Fast Fourier Transform*) inadequate since FFT only transforms data to frequency domain. Another solution is the STFT (*Short-Time Fourier Transform*). However, the crucial drawback is that once a window has been chosen, the time-frequency resolution is fixed over the entire time-frequency plane. In contrast to STFT, the Wavelet Transform uses short windows for high frequencies and long windows for low frequencies. This is a desirable property, especially in analyzing fast transient waveforms such as EEG spikes.

A Wavelet Transform introduces a new mathematical concept to decompose a function $f(t)$ into a set of other functions referred to as wavelet bases:

$$\sum_{\tau,s} = c_{\tau,s}\psi_{\tau,s}(t) \tag{C.4}$$

, where $c_{\tau,s}$ is the wavelet coefficient, $\psi_{\tau,s}(t)$ is the mother wavelet function and is defined as:

$$\psi_{\tau,s}(t) = \frac{1}{\sqrt{s}}\psi(\frac{t-\tau}{s}) \tag{C.5}$$

, where s is the frequency scaling factor, and τ is the translation factor in the time domain.

The DWT applies Wavelet Transform theory with parameters τ and s measured in discrete intervals. It has been proven that any signal that meets the rules of mother wavelet function has a band pass frequency spectrum. This implies that it can be written in the form of filters.

The DWT of a signal x is calculated by passing it through a low-pass filter g and a high-pass filter h. Then the filter outputs are down-sampled by two since half of the frequencies of the signal have been removed. The high-pass filter gives the detail coefficients, and the low-pass filter gives the approximation coefficients. This decomposition has halved the time resolution since only half of each filter output characterizes the signal. However, each output has half of the frequency band of the input signal. Therefore, the frequency resolution has been doubled. This decomposition is repeated to further increase the frequency resolution by decomposing the approximation coefficients by a high-pass and a low-pass filter again. A block diagram of the filter analysis is depicted in Figure C.33.

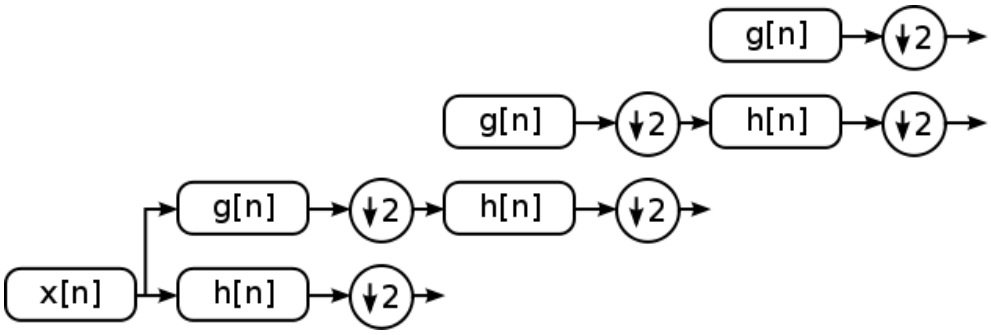


Figure C.33: Block diagram of a 3-level filter analysis in DWT.

The choice of different filter banks gives different DWT. Duabechies-4 DWT has been used quite often on feature extractions for EEG application. For the Daubechies-4 DWT, the filter banks are given as:

$$h(z) = h_0 + h_1 z^{-1} + h_2 z^{-2} + h_3 z^{-3} \tag{C.6}$$

$$g(z) = -h_3 z^2 + h_2 z^1 - h_1 + h_0 z^{-1} \tag{C.7}$$

This leads to the following equations, given x as input sample, s and d are approximation coefficient and detail coefficient correspondingly:

$$s_l^{(1)} = x_{2l} + \sqrt{3}x_{2l+1} \tag{C.8}$$

$$d_l^{(1)} = x_{21+1} - \frac{\sqrt{3}}{4}s_l^{(1)} - \frac{\sqrt{3}-2}{4}s_{l-1}^{(1)} \tag{C.9}$$

$$s_l^{(2)} = s_l^{(1)} + d_{l+1}^{(1)} \tag{C.10}$$

$$S_l = \frac{\sqrt{3}-1}{\sqrt{2}}S_l^{(1)} \tag{C.11}$$

$$d_l = \frac{\sqrt{3}+1}{\sqrt{2}}d_l^{(1)} \tag{C.12}$$

C.7.2. Profiling Information

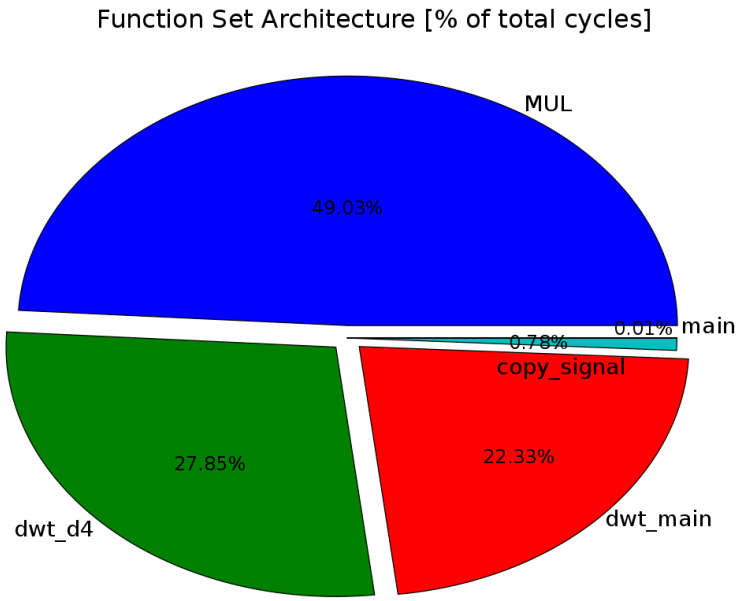


Figure C.34: Function set architecture of the DWT algorithm.

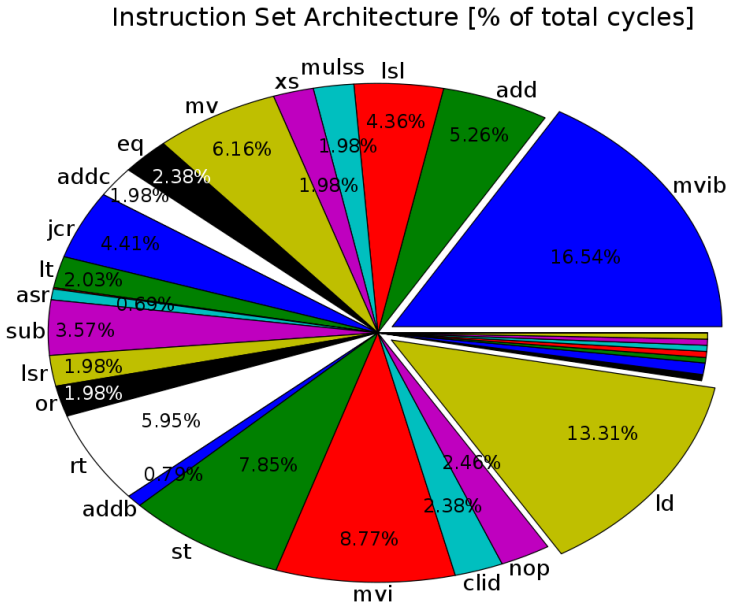


Figure C.35: Instruction set architecture of the DWT algorithm.

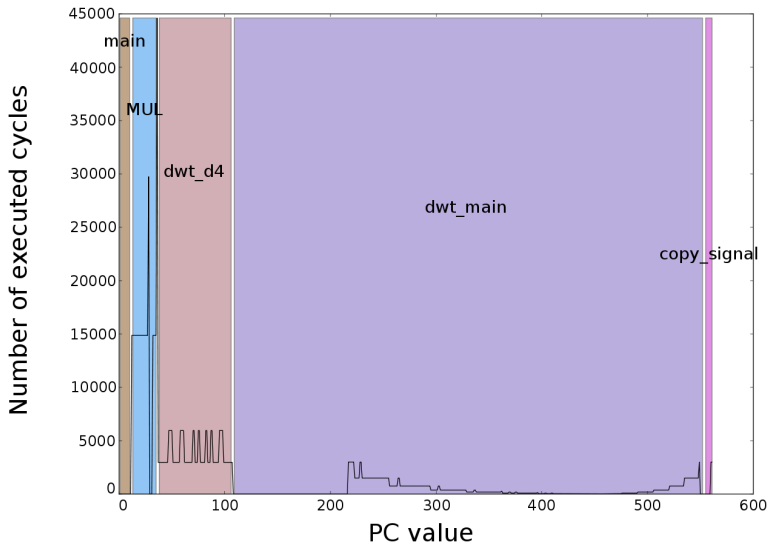


Figure C.36: Program memory footprint of the DWT algorithm.

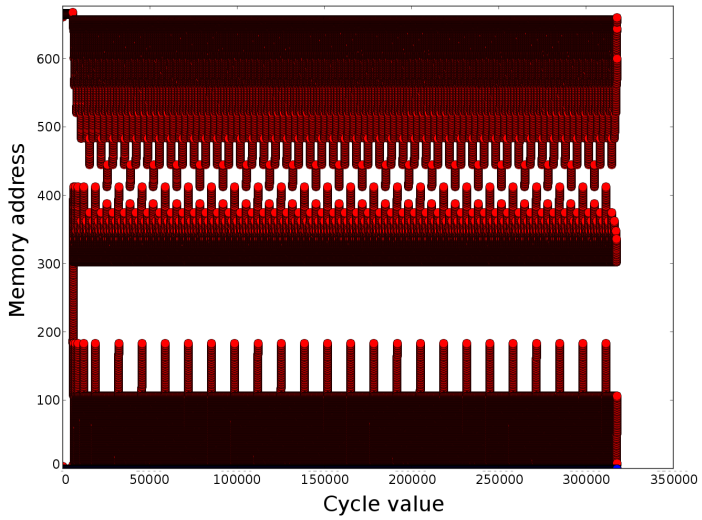


Figure C.37: Profiling information about the access history in the PM of the DWT algorithm.

C.8. Discrete Wavelet Transform Algorithm - Optimised Version

The optimised version of the DWT algorithm is based on the algorithm presented in Section C.7. This Section presents the modifications and the optimisations that are performed not only to implement the optimised version of the DWT algorithm, but also to build the optimised processor architecture over which this optimised algorith is running.

C.8.1. Description of the Algorithm

For the entire signal a a *for* loop can be used to implement Equation C.8, Equation C.9, Equation C.10, Equation C.11, and Equation C.17. However, the amount of memory usage is large if all the EEG data is read at once and feed them to the for loop. This can be solved in the real-time implementation by reading several samples in and producing s and d coefficients at the same time. Thus, it is only needed to store the samples that are read and the output coefficients. However, Equation C.9 and Equation C.10 hinder the real-time implementation because they have references back and forth in time (*i.e.*, S^1_{l-1} and d^1_{l+1}). Fortunately, this can be solved by rewriting the code such that backward reference is removed. In the meanwhile more initialization code and post-periodic code are needed to compensate this modification. The optimized code can be found in [4]. At this stage, it is possible to start explaining the real-time implementation of one-scale DWT. Figure C.38 shows the differences of memory usage of the vector implementation and the real-time implementation for one-scale DWT. Real-time kernel is shown with dashed lines.

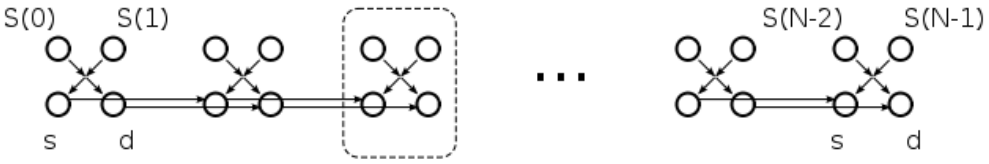


Figure C.38: Memory storage for the vector and the real-time implementation.

For the vector implementation, N input data need to be stored, and $N/2$ s values as well as $N/2$ d values need to be stored. If N is $226,500$ and data type is defined as *short*, there are necessary 906K bytes only for one-scale DWT kernel. While for the real-time implementation, only two input samples and four output data (s and d for the current two samples, as well as s and d from the previous two samples) need to be stored. Therefore, there are only necessary 12 bytes for one-scale DWT kernel.

As mentioned before, DWT has to be performed in several scales in order to reach high frequency resolution. The number of scales depends on the sampling frequency and the resolution requirement. For example, if EEG data is sampled at 256Hz and beta wave activity is considered, which is at the frequency range 8–12 Hz, five scales DWT have to be performed. And if delta wave activity is considered, seven scales DWT have to be performed. Since approximation coefficients are used for the next scale DWT, it is possible to store only one s value from the previous two samples in a buffer and already to start calculating the next scale coefficients using the s value from the current two samples. In order to solve the forward dependencies, the s and the d values for continuous four samples are explicitly stored.

The issue that is omitted so far is the initialization, which needs four samples for each scale. If all the initialization is included for the seven scales, the initialization code size is quite large. However, this can be corrected by shifting the s and the d values each scale by a counter to match the version with vector implementation. Besides, the application contains an inline MUL function which performs a 16 x 16 = 32 multiply with rounding and shifting back to 16 bits.

C.8.2. Profiling Information

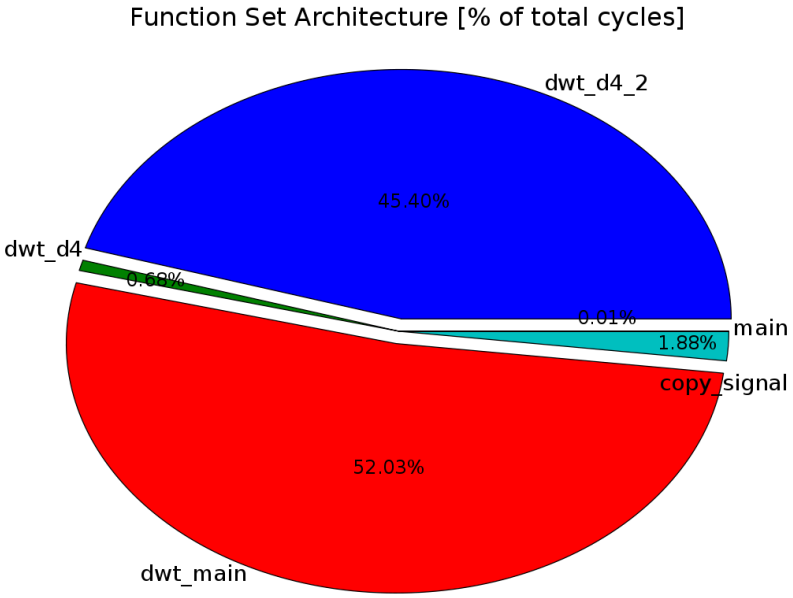


Figure C.39: Function set architecture of the optimised version of the DWT algorithm.

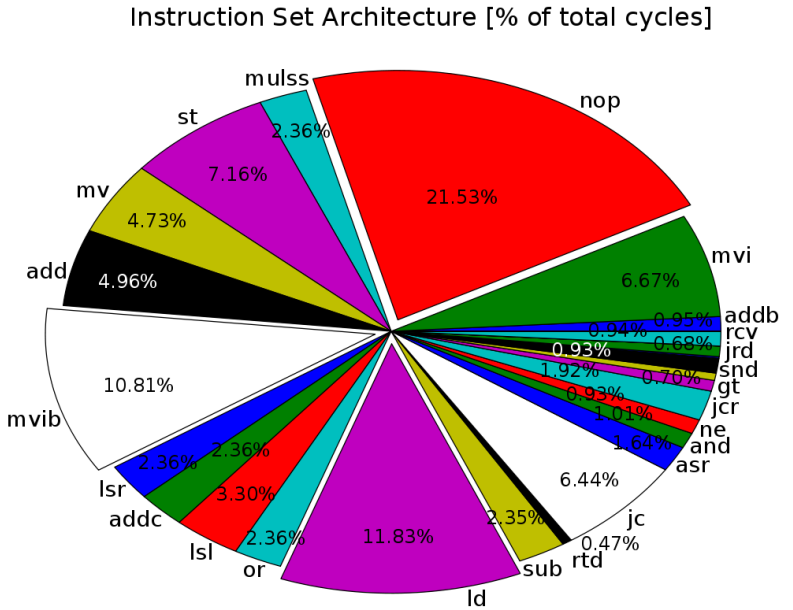


Figure C.40: Instruction set architecture of the optimised version of the DWT algorithm.

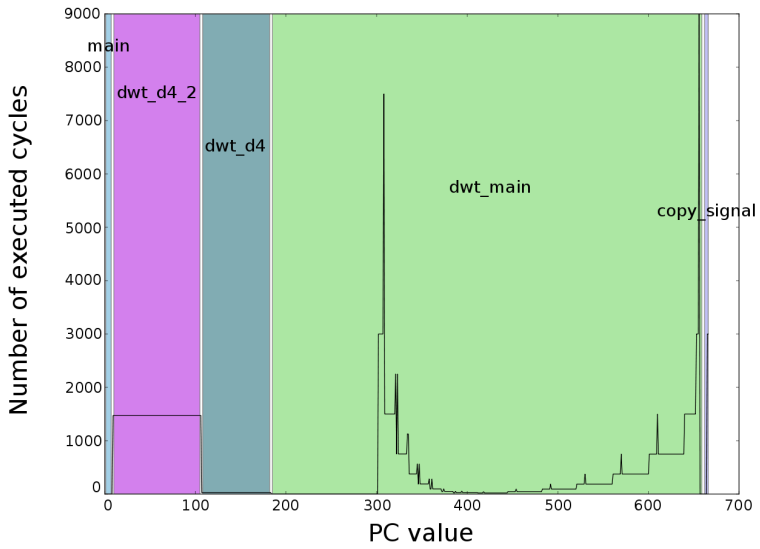


Figure C.41: Program memory footprint of the optimised version of the DWT algorithm.

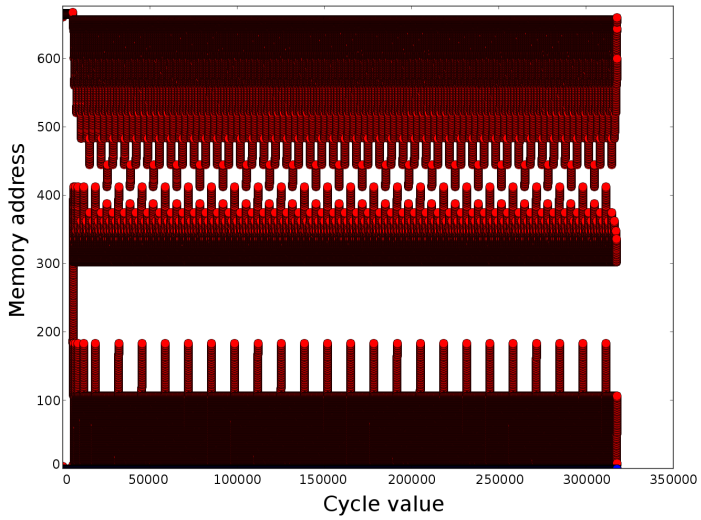


Figure C.42: Profiling information about the access history in the PM of the optimised version of the DWT algorithm.

C.9. Multi-Resolution Frequency Analysis Algorithm

As a complex non-stationary signal, since it was recorded first time, the feature information in EEG signal, which is important for clinical diagnosis and studying, has already been analyzed by various signal analysis methods. With the development of time-frequency analysis methods, many time-frequency analysis methods have also been used to analyze EEG signals and detect the feature. The multi-resolution time-frequency analysis method based on STFT (*Short-Time Fourier Transform*) and wavelet packet transform has been implemented to advance the self-adaptive ability for signals, so more flexible division of frequency bands in EEG is obtained and the basic rhythms in EEG signals can be detected efficiently.

C.9.1. Description of the Algorithm

The Gabor Transform of a signal $x(t)$ is defined as follows:

$$g_D(f,t) = \int_{-\infty}^{+\infty} x(t') g_D^*(t'-t) e^{-j2\pi f t'} dt \tag{C.13}$$

, where $*$ denotes complex conjugation.

The window function g_D is introduced in order to localize the Fourier Transform of the signal at time t. So, the window function must be peaked around t and falling off rapidly. There are several window functions (*e.g.*, Hanning window function, Hamming window function, Gaussian window function, etc.) that can be used to achieve this goal. Among those, Gabor Transform proposed to use a Gaussian window function:

$$g_a(t) = (\frac{\alpha}{\pi})^{1/4} e^{-\frac{\alpha}{2}t^2} \tag{C.14}$$

Due to the fact that the Fourier Transform of a Gaussian function is still a Gaussian function, this allows a simultaneous localization in time domain and frequency domain.

Because of the using of fixed window, Gabor transform (or STFT) is of its inherent limitation, but it is still a good time-frequency analysis method for many applications. Firstly, it has no cross-term. Secondly, the localization of the signal can be obtained by adding the window, and it is of intuitionistic

physical meanings and fast algorithm based on Fourier Transform. Because of the introduction of multi-resolution analysis, Wavelet Transform with Mallat algorithm can decompose a signal orthogonally into multi scales signal's components, which is of great convenience for understanding a signal in time domain. However, the information in time domain is not straightforward, even the time-frequency spectrum cannot be identified. Considering the excellence and the shortcoming of Wavelet Transform with flexible window and STFT with fixed window, it is possible to combine them together to perform the time-frequency analysis of EEG signals. The steps of multi-resolution time-frequency analysis are shown as the following:

- Decompose $s(t)$ into M signal components with different scales by using Mallat algorithm:

$$s(t) = \sum_{m=0}^{M-1} s_m(t) \tag{C.15}$$

- Do STFT with window function $h_m(t)$ with right window width for the signal components $s_m(t)$ obtained by decomposition:

$$s_{m,\tau}(\omega) = \int_{-\infty}^{+\infty} e^{j\omega t} S_m(\tau) h(\tau - t) d\tau \tag{C.16}$$

- Add time-frequency distributions of each scaled signals in the same time-frequency plane to obtain multi-resolution time-frequency analysis.

$$s_t(\omega) = \sum_{m=0}^{M-1} s_{m,\tau}(\omega) \tag{C.17}$$

Figure C.43 shows the flowchart of the MRFA (*Multi-Resolution Frequency Analysis*) algorithm.

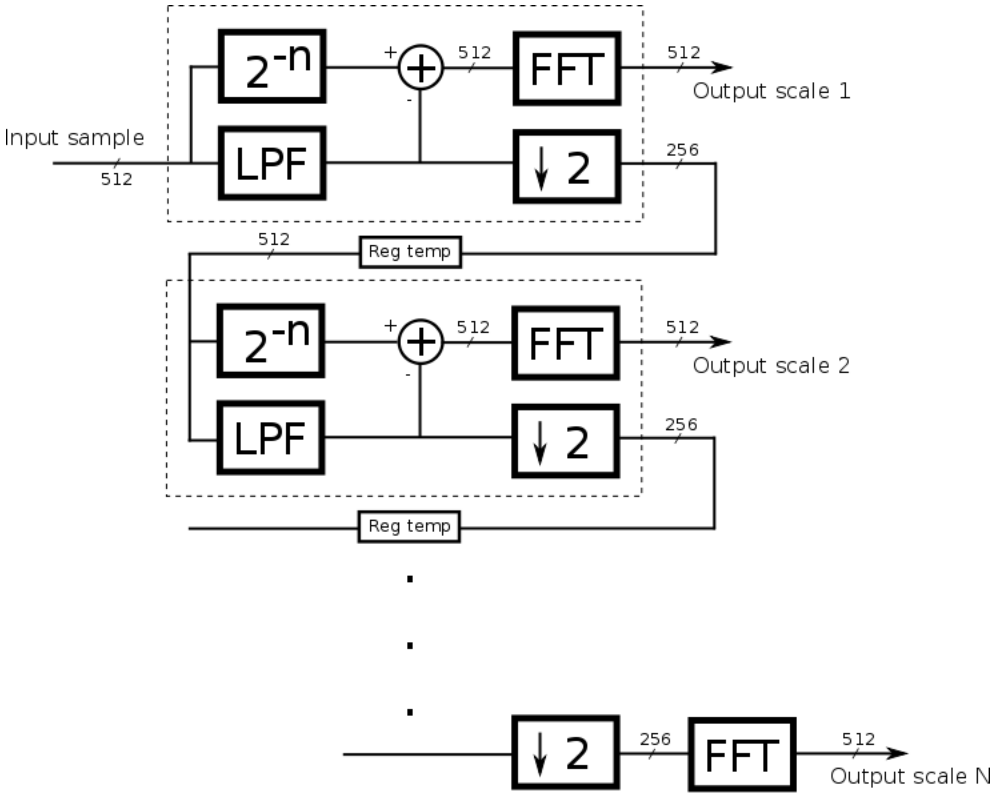


Figure C.43: Flowchart of the MRFA algorithm.

C.9.2. Profiling Information

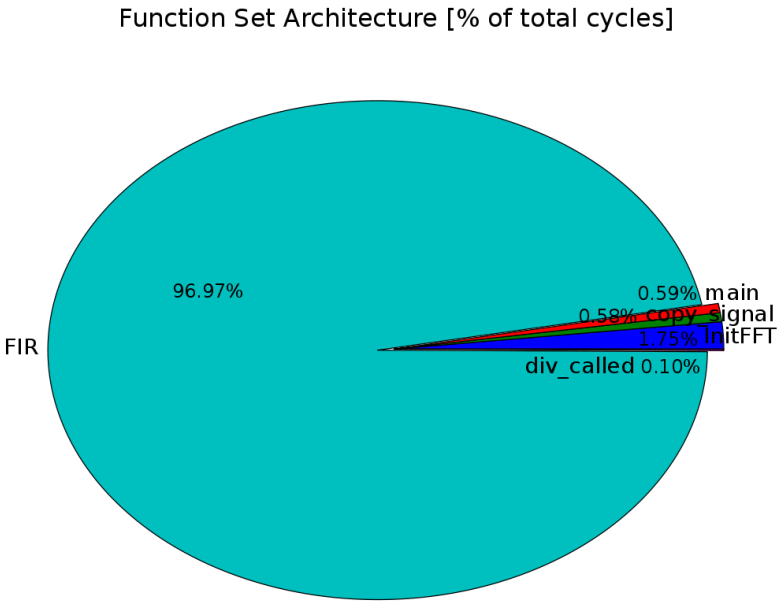


Figure C.44: Function set architecture of the MRFA algorithm.

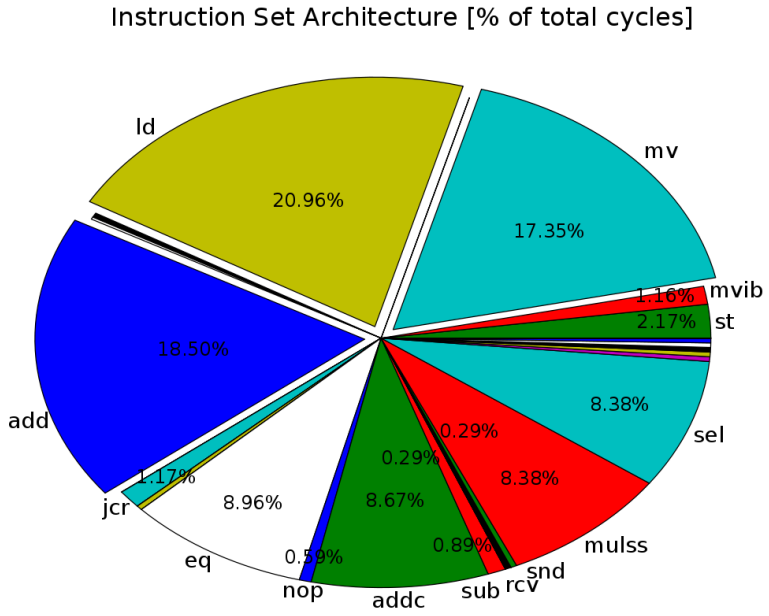


Figure C.45: Instruction set architecture of the MRFA algorithm.

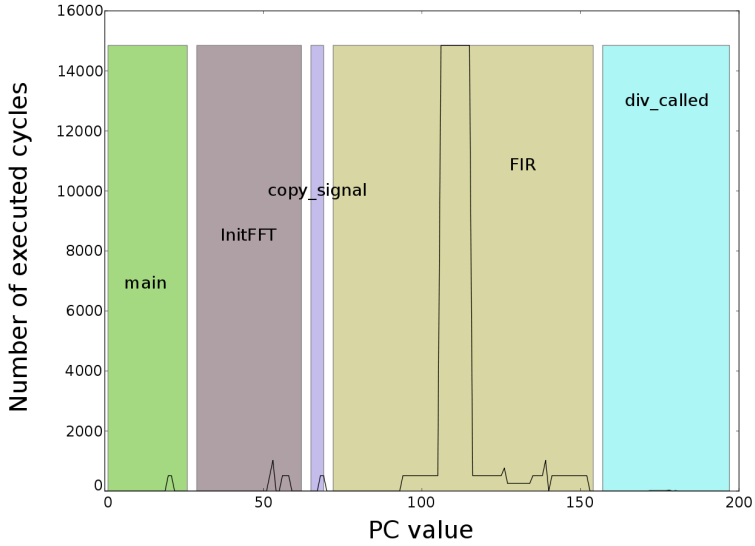


Figure C.46: Program memory footprint of the MRFA algorithm.

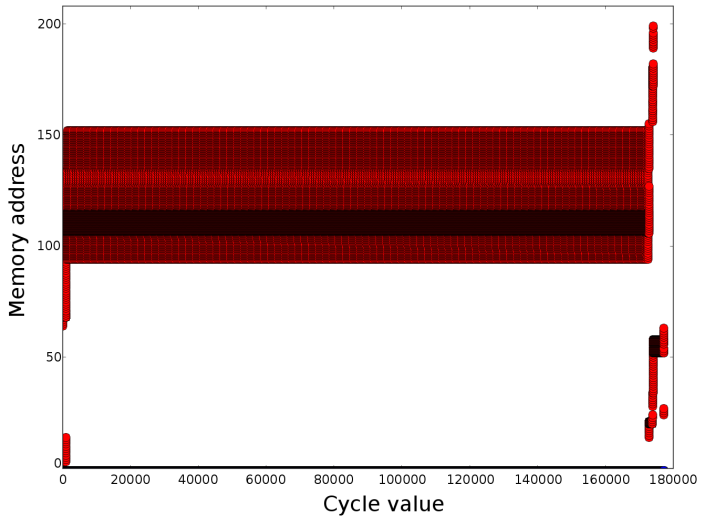


Figure C.47: Profiling information about the access history in the PM of the MRFA algorithm.

Bibliography

[Ace02] Oscar Acevedo. A survey of software optimization techniques for low-power consumption. Technical Report Computing Research Conference CRC2002, University of Puerto Rico, Mayagüez Campus, Puerto Rico 00680, December 2002.

[AG09] Cesare Alippi and Cristian Galperti. Energy storage mechanisms in low power embedded systems: Twin batteries and supercapacitors. In *International Conference on Wireless Communication, Vehicular Technology, Information Theory and Aerospace Electronic Systems Technology*, pages 31–35, May 2009.

[AJB+03] Tom Vander Aa, Murali Jayapala, Francisco Barat, Henk Corporaal, Francky Catthoor, and Geert Deconinck. Software transformations to reduce instruction memory power consumption using a loop buffer. Technical Report Code Generation and Optimization, Departement Elektrotechniek - ESAT, Leuven, Belgium, January 2003.

[ANMD07] Rabie Ben Atitallah, Smaïl Niar, Samy Meftali, and Jean-Luc Dekeyser. Mpsoc power estimation framework at transaction level modeling. In *Internatonal Conference on Microelectronics*, pages 245–248, December 2007.

[ARM12] ARM Website, 2012. Available online: http://www.arm.com/ (Accessed on 18th September 2012).

[BBB+05] Luca Benini, Davide Bertozzi, Alessandro Bogliolo, Francesco Menichelli, and Mauro Olivieri. Mparm: Exploring the multi-processor soc design space with systemc. *Journal of VLSI Signal Processing Systems*, 41(2):169–182, September 2005.

[BCDV09] Chiara Buratti, Andrea Conti, Davide Dardari, and Roberto Verdone. An overview on wireless sensor networks technology and evolution. *MDPI Sensors*, 9(9):1–28, August 2009.

[BHK+97] Raminder S. Bajwa, Mitsuru Hiraki, Hirotsugu Kojima, Douglas J. Gorny, Kenichi Nitta, Avadhani Shridhar, Koichi Seki, and Katsuro Sasaki. Instruction buffering to reduce power in processors for signal processing. *IEEE Transactions on Very Large Scale Integration Systems*, 5(4):417–424, December 1997.

[BHPS99] Nikolaos Bellas, Ibrahim N. Hajj, Constantine D. Polychronopoulos, and George D. Stamoulis. Energy and performance improvements in microprocessor design using a loop cache. In *Proceedings of the IEEE International Conference on Computer Design*, pages 378–383, Washington, United States of America, October 1999. IEEE Computer Society.

[BLRC05] Nikhil Bansal, Kanishka Lahiri, Anand Raghunathan, and Srimat T. Chakradhar. Power monitors: A framework for system-level power estimation using heterogeneous power models. In *Proceedings of the International Conference on VLSI Design held jointly with International Conference on Embedded Systems Design*, pages 579–585, Washington, United States of America, January 2005. IEEE Computer Society.

[BMP00] Luca Benini, Alberto Macii, and Massimo Poncino. A recursive algorithm for low-power memory partitioning. In *Proceedings of the International Symposium on Low-Power Electronics and Design*, pages 78–83, New York, United States of America, July 2000. ACM.

[BMP03] Luca Benini, Alberto Macii, and Massimo Poncino. Energy-aware design of embedded memories: A survey of technologies, architectures, and optimization techniques. *ACM Transactions of Embedded Computing System*, 2(1):5–32, February 2003.

[BSBD+08] David Black-Schaffer, James Balfour, William Dally, Vishal Parikh, and JongSoo Park. Hierarchical instruction register organization. *IEEE Computer Architecture Letters*, 7(2):41–44, July 2008.

[BSL+02] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad memory: Design alternative for cache on-chip memory in embedded systems. In *Proceedings of the tenth international symposium on Hardware/software codesign*, pages 73–78, New York, United States of America, May 2002. ACM.

[CAD12] Cadence Design System Website, 2012. Available online: http://www.cadence.com/us/pages/default.aspx (Accessed on 18th September 2012).

[CBW+09] Steven C.Jocke, Jonathan F. Bolus, Stuart N. Wooters, Travis N. Blalock, and Benton H. Calhoun. A 2.6 uw sub-threshold mixed-signal ecg soc. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, pages 117–118, New York, United States of America, August 2009. ACM.

[CC08] S. Chalasani and J.M. Conrad. A survey of energy harvesting sources for embedded systems. In *Proceedings of IEEE Southeast conference*, pages 442–447, Washington, United States of America, April 2008. IEEE Computer Society.

[CRL+10] Francky Catthoor, Praveen Raghavan, Andy Lambrechts, Murali Jayapala, Angeliki Kritikakou, and Javed Absar. *Ultra-Low Energy Domain-Specific Instruction-Set Processors*. Springer Publishing Company, Incorporated, Dordrecht, The Netherlands, 2010.

[CY02] Wissam Chedid and Chansu Yu. Survey on power management techniques for energy efficient computer systems. Technical Report CSU-ECE-TR-02-01, Cleveland State University, Cleveland, United States of America, 2002.

[dAEES97] Edwin de Angel and Jr. Earl E. Swartzlander. Survey of low power techniques for roms. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 7–11, New York, United States of America, August 1997. ACM.

[DS98] Ingrid Daubechies and Wim Sweldens. Factoring wavelet transforms into lifting steps. *Journal of Fourier Analysis and Applications*, 4(3):247–269, March 1998.

[Dwo04] M. Dworkin. *Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality*. National Institute of Standars and Technology (NIST) - Special Publication 800-38C, Gaithersburg, United States of America, 2004.

[FEL01] Xiaobo Fan, Carla Ellis, and Alvin Lebeck. Memory controller policies for dram power management. In *Proceedings of the International Symposium on Low-Power Electronics and Design*, pages 129–134, New York, United States of America, August 2001. ACM.

[GAG+00] A. L. Goldberger, L. A. N. Amaral, L. Glass, J. M. Hausdorff, P. Ch. Ivanov, R. G. Mark, J. E. Mietus, G. B. Moody, C.-K. Peng, and H. E. Stanley. Physiobank, physiotoolkit, and physionet: Components of a new research resource for complex physiologic signals. *Circulation*, 101(23):215–220, June 2000.

[GG04] Arijit Ghosh and Tony Givargis. Cache optimization for embedded processor cores: An analytical approach. *ACM Transactions on Design Automation of Electronic Systems*, 9(4):419–440, October 2004.

[GLW05] Zhiguo Ge, H. B. Lim, and Weng Fai Wong. Memory hierarchy hardware-software co-design in embedded systems. Technical Report Computer Science, Massachusetts Institute of Technology, Cambridge, United States of America, January 2005.

[GMV+04] Jose Ignacio Gomez, Paul Marchal, Sven Verdoorlaege, Luis Pinuel, and Francky Catthoor. Optimizing the memory bandwidth with loop morphing. In *IEEE International Conference Proceedings of the Application-Specific Systems, Architectures and Processors*, pages 213–223, Washington, DC, USA, September 2004. IEEE Computer Society.

[GOO12] Google Website, 2012. Available online: http://www.google.com (Accessed on 18th September 2012).

[GRVD09] Ann Gordon-Ross, Frank Vahid, and Nikil Dutt. Fast configurable-cache tuning with a unified second-level cache. *IEEE Transactions on Very Large Scale Integration Systems*, 17(1):80–91, January 2009.

[HP07] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, San Francisco, United States of America, 2007.

[IBM12] IBM Website, 2012. Available online: http://www.ibm.com/us/en/ (Accessed on 18th September 2012).

[IEE12] IEEE Standards Association, 2012. IEEE Standards Association - $IEEE$ 802.15^{TM}: Wireless Personal Area Networks (PANs). Available online: http://standards.ieee.org/about/get/802/802.15.html (Accessed on 18th September 2012).

[IMM02] Koji Inoue, Vasily G. Moshnyaga, and Kazuaki Murakami. A history-based i-cache for low-energy multimedia applications. In *Proceedings of the 2002 International Symposium on Low Power Electronics and Design*, pages 148–153, New York, United States of America, August 2002. ACM.

[INT11] Intel Website, 2011. Available online: http://www.intel.es/content/www/es/es/homepage.html (Accessed on 18th September 2012).

[ITR12] International Technology Roadmap for Semiconductors Website, 2012. Available online: http://www.itrs.net/ (Accessed on 18th September 2012).

[IY00] Tohru Ishihara and Hiroto Yasuura. A power reduction technique with object code merging for application specific embedded processors. In *Proceedings of the Conference on Design, Automation, and Test in Europe*, pages 617–623, New York, United States of America, Marzo 2000. ACM.

[JBA+05] Murali Jayapala, Francisco Barat, Tom Vander Aa, Francky Catthoor, Henk Corporaal, and Geert Deconinck. Clustered loop buffer organization for low energy vliw embedded processors. *IEEE Transactions on Computers*, 54(6):672–683, June 2005.

[JBB+02] Murali Jayapala, Francisco Barat, Pieter Op De Beeck, Rudy Lauwereins, Francky Catthoor, and Geert Deconinck. A low energy clustered instruction memory hierarchy for long instruction word processors. In *Proceedings of the International Workshop on Power And Timing Modeling, Optimization and Simulation*, pages 258–267, London, United Kingdom, September 2002. Springer-Verlag.

[Jou90] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, New York, United States of America, May 1990. ACM.

[KAA+07] Felipe Klein, Guido Araujo, Rodolfo Azevedo, Roberto Leao, and Luiz C. V. dos Santos. An efficient framework for high-level power exploration. In *Midwest Symposium on Circuits and Systems*, pages 1046–1049, Washington, United States of America, August 2007. IEEE Computer Society.

[KGMS97] Johnson Kin, Munish Gupta, and William H. Mangione-Smith. The filter cache: an energy efficient memory structure. In

Proceedings of the ACM/IEEE International Symposium on Microarchitecture, pages 184–193, Washington, United States of America, December 1997. IEEE Computer Society.

[KKC+04] Mahmut Kandemir, Ismail Kadayif, Alok Choudhary, J. Ramanujam, and Ibahim Kolcu. Compiler-directed scratch pad memory optimization for embedded multiprocessors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(3):281–287, March 2004.

[KKK02] Mahmut T. Kandemir, Ibrahim Kolcu, and Ismail Kadayif. Influence of loop optimizations on energy consumption of multi-bank memory systems. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 276–292, April 2002.

[KYS+11] Hyejung Kim, Refet Firat Yazicioglu, Kim Sunyoung, Nick Van Helleputte, Antonio Artes, Mario Konijnenburg, Jos Huisken, Julien Penders, and Chris Van Hoof. A configurable and low-power mixed signal soc for portable ecg monitoring applications. In *Symposium on VLSI Circuits (VLSIC)*, pages 142 –143, Washington, United States of America, June 2011. IEEE Computer Society.

[Lam09] Andy Lambrechts. *Energy-Aware Datapath Optimizations at the Architecture-Compiler Interface*. PhD thesis, Departement Elektrotechniek - ESAT, Faculty of Electrical Engineering, K. U. Leuven, 2009.

[LAR+05] I. Romero Legarreta, Paul S. Addison, M. J. Reed, Neil R. Grubb, Gareth R. Clegg, Colin E. Robertson, and James N. Watson. Continuous wavelet transform modulus maxima analysis of the electrocardiogram: Beat characterisation and beat-to-beat measurement. *International Journal of Wavelets, Multiresolution and Information Processing*, 3(1):19–42, March 2005.

[LCL+07] Fabien Lotte, Marco Congedo, Anatole Lécuyer, Fabrice Lamarche, and Bruno Arnaldi. A review of classification algorithms for eeg-based brain-computer interfaces. *Journal of Neural Engineering*, 4(2):1–24, June 2007.

[LK04] Chun-Gi Lyuh and Taewhan Kim. Memory access scheduling and binding considering energy minimization in multi-bank memory systems. In *Proceedings of the annual Design Automation Conference*, pages 81–86, New York, United States of America, June 2004. ACM.

[LKY+06] Ikhwan Lee, Hyunsuk Kim, Peng Yang, Sungjoo Yoo, Eui-Young Chung, Kyu-Myung Choi, Jeong-Taek Kong, and Soo-Kwan Eo. Powervip: Soc power estimation framework at transaction level. In *Proceedings of the Asia and South Pacific Design Automation Conference*, pages 551–558, Piscataway, United States of America, January 2006. IEEE Press.

[LMA99] Lea Hwang Lee, Bill Moyer, and John Arends. Instruction fetch energy reduction using loop caches for embedded applications with small tight loops. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 267–269, New York, United States of America, August 1999. ACM.

[LVB+06] Toon Leroy, Erik Vranken, Andres Van Brecht, E. Struelens, B. Sonck, and Daniel Berckmans. A computer vision method for on-line behavioral quantification of individually caged poultry. *Transactions of the American Society of Agricultural and Biological Engineers*, 49(3):795–802, May 2006.

[LZSG02] Nikolaos D. Liveris, Nikolaos D. Zervas, Dimitrios Soudris, and Constantinos E. Goutis. A code transformation-based methodology for improving i-cache performance of dsp applications. In *Proceedings of the Conference on Design, Automation, and Test in Europe*, pages 977–983, Washington, United States of America, March 2002. IEEE Computer Society.

[Man05] Hugo De Man. Ambient intelligence: gigascale dreams and nanoscale realities. In *IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, pages 29–35, Washington, DC, USA, February 2005. IEEE Computer Society.

[Mar00] Diana Marculescu. Profile-driven code execution for low power dissipation. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 253–255, New York, United States of America, 2000. ACM.

[MAZA05] Itziar Marin, Eduardo Arceredillo, Aitzol Zuloaga, and Jagoba Arias. Wireless sensor networks: A survey on ultra-low power-aware design. *Proceedings of World Academy of Science, Engineering, and Technology*, 8:44–49, October 2005.

[NIS12] Advanced Encryption Standard (AES), 2012. National Institute of Standards and Technology (NIST) - FIPS PUBS 197. Available online: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf (Accessed on 18th September 2012).

[PCD+01] Preeti Ranjan Panda, Francky Catthoor, Nikil D. Dutt, Koen Danckaert, Erik Brockmeyer, Chidamber Kulkarni, Arnout Vandecappelle, and Per Gunnar Kjeldsberg. Data and memory optimization techniques for embedded systems. *ACM Transactions on Design, Automotation of Electronics Systems*, 6(2):149–206, April 2001.

[PFH+12] Georgia Psychou, Robert Fasthuber, Jos Hulzink, Jos Huisken, and Francky Catthoor. Sub-word handling in data-parallel mapping. In *Architecture of Computing Systems Workshops*, pages 1–7, Washington, United States of America, February 2012. IEEE Computer Society.

[PH90] Karl Pettis and Robert C. Hansen. Profile guided code positioning. *ACM SIGPLAN Notices*, 25(6):16–27, June 1990.

[PV97] Massoud Pedram and Hirendu Vaishnav. Power optimization in vlsi layout: A survey. *Journal of VLSI Signal Processing Systems*, 15(3):221–232, March 1997.

[QJ04] Shuren Qin and Zhong Ji. Multi-resolution time-frequency analysis for detection of rhythms of eeg signals. In *Digital Signal Processing Workshop and the IEEE Signal Processing Education Workshop*, pages 338–341, Washington, United States of America, August 2004. IEEE Computer Society.

[RB99] Raghuveer M. Rao and Ajit S. Bopardikar. *Wavelet Transforms: Introduction to Theory and Applications*. Prentice Hall PTR, Pennsylvania, United States of America, 1999.

[RLJ+06] Praveen Raghavan, Andy Lambrechts, Murali Jayapala, Francky Catthoor, and Diederik Verkest. Distributed loop controller architecture for multi-threading in uni-threaded vliw processors. In *Proceedings of the conference on Design, automation and test in Europe*, pages 339–344, Leuven, Belgium, March 2006. European Design and Automation Association.

[Sch02] John P. Scheible. A survey of storage options. *Computer*, 35(12):42–46, December 2002.

[SCL+04] Eugene Shih, Seonghwan Cho, Fred S. Lee, Benton H. Calhoun, and Anantha Chandrakasan. Design considerations for energy-efficient radios in wireless microsensor networks. *Journal of VLSI Signal Processing Systems*, 37(1):77–94, May 2004.

[SGW01] Loren Schwiebert, Sandeep K.S. Gupta, and Jennifer Weinmann. Research challenges in wireless networks of biomedical sensors. In *Proceedings of the Annual International Conference on Mobile Computing and Networking*, pages 151–165, New York, NY, USA, July 2001. ACM.

[SPE12] Standard Performance Evaluation Corporation Website, 2012. Available online: http://www.spec.org/benchmarks.html (Accessed on 18th September 2012).

[SUN12] Sun Microsystems in Oracle Website, 2012. Available online: http://www.oracle.com/us/sun/index.htm (Accessed on 18th September 2012).

[SWKS01] Ayad Salhieh, Jennifer Weinmann, Manish Kochhal, and Loren Schwiebert. Power efficient topologies for wireless sensor networks. In *International Conference on Parallel Processing*, pages 156–163, September 2001.

[SYN12] Synopsys Website, 2012. Available online: http://www.synopsys.com/home.aspx (Accessed on 18th September 2012).

[TAR12] Target Website, 2012. Available online: http://www.retarget.com/ (Accessed on 18th September 2012).

[TGN02] Weiyu Tang, Rajesh Kumar Gupta, and Alexandru Nicolau. Power savings in embedded processors through decode filer cache. In *Proceedings of the conference on Design, automation and test in Europe*, pages 443–448, Washington, United States of America, March 2002. IEEE Computer Society.

[TSH+10] I. Tsekoura, G. Selimis, J. Hulzink, F. Catthoor, J. Huisken, H. de Groot, and C. Goutis. Exploration of cryptographic asip designs for wireless sensor designs. In *Proceedings of the IEEE International Conference on Electronics, Circuits, and Systems*, pages 827–830, Washington, United States of America, November 2010. IEEE Computer Society.

[Uni12] Arm, 2012. Advanced RISC Machines Ltd. - ARM7TDMI Reference Manual. Available online: http://www.arm.com/pdfs/DDI0210B_7TDMI_R4.pdf (Accessed on 18th September 2012).

[VIR12] Virage Memories in Synopsys Website, 2012. Available online: http://www.synopsys.com/IP/SRAMandLibraries/Pages/default.aspx (Accessed on 18th September 2012).

[VLCV01] Jason Villarreal, Roman Lysecky, Susan Cotterell, and Frank Vahid. A study on the loop behavior of embedded programs. Technical Report UCR-CSE-01-03, University of California, Riverside, United States of America, December 2001.

[VM07] Manish Verma and Peter Marwedel. *Advanced Memory Optimization Techniques for Low-Power Embedded Processors*. Springer Publishing Company, Incorporated, Dordrecht, The Netherlands, 2007.

[VSB04] Kugan Vivekanandarajah, Thambipillai Srikanthan, and Saurav Bhattacharyya. Dynamic filter cache for low power instruction memory hierarchy. In *Proceedings of the Euromicro Symposium on Digital System Design*, pages 607–610, Washington, United States of America, August 2004. IEEE Computer Society.

[Wel95] Gregory F. Welch. A survey of power management techniques in mobile computing operating systems. *ACM SIGOPS Operating Systems Review*, 29(4):47–56, October 1995.

[WK03] Wayne Wolf and Mahmut T. Kandemir. Memory system optimization of embedded software. *Proceedings of the IEEE*, 91(1), January 2003.

[WLLP01] Brett Warneke, Matt Last, Brian Liebowitz, and Kristofer S. J. Pister. Smart dust: Communicating with a cubic-millimeter computer. *Computer*, 34(1):44–51, January 2001.

[WMK+08] Alan C-W. Wong, Declan McDonagh, Ganesh Kathiresan, Okundu C. Omeni, Omar El-Jamaly, Thomas C-K. Chan, Paul Paddan, and Alison J. Burdett. A 1v, micropower system-on-chip for vital-sign monitoring in wireless body sensor networks. In *IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 138–139, Washington, United States of America, February 2008. IEEE Computer Society.

[YKH+09] Y.H. Yassin, P.G. Kjeldsberg, J. Hulzink, I. Romero, and J. Huisken. Ultra low-power application specific instruction-set processor design for a cardiac beat detector algorithm. In *Proceedings of the NORCHIP conference*, pages 1–4, Washington, United States of America, November 2009. IEEE Computer Society.

[YKT+10] Refet Firat Yazicioglu, Sunyoung Kim, Tom Torfs, Patrick Merken, and Chris Van Hoof. A 30 uw analog signal processor

asic for biomedical signal monitoring. In *IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 124–125, Washington, United States of America, February 2010. IEEE Computer Society.

[YQ04] Shi Yunhui and Ruan Qiuqi. Continuous wavelet transforms. In *Proceedings of the International Conference on Signal Processing*, pages 207–210, Washington, United States of America, September 2004. IEEE Computer Society.

[ZFMS05] Hongtao Zhong, Kevin Fan, Scott Mahlke, and Michael Schlansker. A distributed control path architecture for vliw processors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 197–206, Washington, United States of America, September 2005. IEEE Computer Society.

[Zha05] Chuanjun Zhang. An efficient direct mapped instruction cache for application-specific embedded systems. In *Proceedings of the third IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 45–50, New York, United States of America, September 2005. ACM.

[ZLM07] Hongtao Zhong, Steven A. Lieberman, and Scott A. Mahlke. Extending multicore architectures to exploit hybrid parallelism in single-thread applications. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, pages 25–36, Washington, United States of America, February 2007. IEEE Computer Society.

[ZXL07] Qiang Zhou, Huagang Xiong, and Hengqing Lin. Real-time performance analysis for wireless sensor networks. In *Proceedings of the IFIP International Conference on Network and Parallel Computing Workshops*, pages 337–342, Washington, United States of America, September 2007. IEEE Computer Society.

"Y así, del mucho leer y del poco dormir, se le secó el celebro de manera que vino a perder el juicio."

El ingenioso hidalgo don Quijote de la Mancha.
— Miguel de Cervantes Saavedra.

"Science has not yet taught us if madness is or is not the sublimity of the intelligence."

— Edgar Allan Poe.

"All truth is simple... Is that not doubly a lie?"

— Friedrich Wilhelm Nietzsche.